technische universität
dortmund

Masterarbeit

Modellgetriebenes Reengineering der Geschäftslogik von Java-Applikationen

Vera Tomskikh

Erstgutachter	Prof. Dr. Bernhard Steffen
Zweitgutachter	Dr. Andreas Wagener
Betreuer	Dipl.-Inf. Stefan Naujokat

Technische Universität Dortmund
Fakultät für Informatik

In Kooperation mit
Opitz Consulting Deutschland GmbH

Dezember 2013

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In vielen Unternehmen werden Geschäftsprozesse durch Software abgebildet oder unterstützt. Oft werden Prozesse einmalig spezifiziert und im Programmcode umgesetzt. In der Zeit von wachsenden fachlichen und technologischen Ansprüchen an Software-Systeme, müssen diese ständig angepasst werden. Durch kurzfristige Änderungen und häufige Erweiterungen entstehen die sogenannten Legacy-Systeme, die sich oft durch Unübersichtlichtlichkeit, abwesende Dokumentation und schwere Wartbarkeit auszeichnen. Viele dieser Systeme sind zudem durch den Programmierstil, durch einen Mangel an Dokumentation und Verwendung teilweise undurchsichtiger Optimierungen schwer verständlich und daher auch aufwändig zu warten und ändern. Hinzu kommt, dass adaptive und kurzfristige Anpassungen die Struktur der Systeme zerstören können und oftmals redundanten Code erzeugen. Die Einarbeitung zusätzlicher Entwickler beansprucht viel Zeit, besonders wenn die Dokumentation fehlt oder veraltet ist. Falls die Dokumentation existiert, muss diese bei jeder Prozessänderung angeglichen werden. Durch unflexible Struktur und lückenhafte Dokumentation können bei jeder Änderung des Programmcodes fachliche Fehler verursacht werden.

Die Modernisierung von Altsystemen stellt sich heutzutage als eine regelmäßig zu lösende Aufgabe dar [47]. In der Literatur werden drei Ansätze im Umgang mit solchen Altsystemen diskutiert: Reimplementierung, Ablösung durch Standard-Komponenten und Reengineering. Durch eine Neuimplementierung kann die Qualität der Software deutlich verbessert werden. Allerdings ist diese Strategie oft mit hohen Kosten und großem Aufwand verbunden. Ein Umstieg auf Standard-Software ist nur dann möglich, wenn die Geschäftslogik gut dokumentiert ist und ohne große Anpassungen durch die gewählte Software abgebildet werden kann.

Die dritte Möglichkeit umfasst das *Reengineering* von Legacy-Systemen. Reengineering ist der Oberbegriff für alle Prozesse, deren Ziel die qualitative Verbesserung der Struktur von Software ist. Im Vergleich zu anderen Strategien im Umgang mit Legacy-Systemen weist Reengineering mehrere Vorteile auf. Einige Teile des Reengineering-Prozesses können automatisiert werden. Geschäftsprozesse können in der Software nachgebildet werden, auch wenn diese komplex sind und von Standards abweichen. Beim Reengineering wird das System zunächst in eine höhere Abstraktionsebene überführt, um Zusammenhänge zwischen existierenden Komponenten zu rekonstruieren. In diesem Schritt werden die Elemente der Software und deren Beziehungen mit Hilfe von *Modellen* dargestellt. Modelle dienen zur Abbildung eines Objektes mittels einer vordefinierten domänenspezifischen Sprache. Durch ihre Abstraktionseigenschaften helfen sie, die verlorene Übersichtlichkeit und Klarheit des

Software-Systems wiederherzustellen. Im nächsten Schritt werden die entstandene Modelle analysiert und angepasst, sodass diese die Geschäftsprozesse möglichst genau abbilden. Darauf folgend wird das System basierend auf anderen Technologien bzw. in einer anderen Programmiersprache erneut implementiert. Beim Reengineering wird lediglich die Implementierung verändert, das Verhalten des Systems bleibt erhalten.

Eine besondere Form stellt das sogenannte *modellgetriebene* Reengineering dar, bei dem die Modelle nach der Extraktion zu den Hauptartefakten des neuen Systems werden [46]. Ähnlich wie bei der modellgetriebenen Entwicklung wird aus fertigen Modellen neuer Quellcode generiert. Als Ergebnis des Reengineerings entsteht ein System, welches modellgetrieben weiterentwickelt werden kann. Es existieren mehrere Gründe, warum modellgetriebenes Reengineering vorteilhaft sein kann. Die Modelle dienen nicht nur der Abstraktion, diese beinhalten die Dokumentation der Prozesse und sind unabhägig von den technischen Implementierungsdetails. Das System nach dem Reengineering besitzt eine wohlgeformte Struktur und eine einheitliche Architektur. Die Modelle sind übersichtlich und die Anwendungslogik ist dadurch leichter zu verstehen. Fachexperte können bei der Modellierung nicht nur unterstützen, sondern aktiv daran teilnehmen. Dadurch können Fehlinterpretationen der Prozesse frühzeitig korrigiert werden.

Die größte Schwierigkeit des modellgetriebenen Reengineerings besteht darin, die oftmals lediglich im Quellcode existierende Geschäftslogik in Modellen korrekt abzubilden. Mit Hilfe geeigneter Werkzeuge können Modelle per Hand erstellt und schrittweise verfeinert werden. Bei großen Systemen ist eine manuelle Modellbildung oft leider nicht handhabbar. Alternativ dazu kann man bestimmte Quellcode-Elemente (wie z.B. Methoden oder Anweisungen in Methoden) automatisch in Modellelemente transformieren. Diese feingranularen Einheiten werden dann in mehreren Iterationen zu abstrakteren Modellen zusammengefasst, bis sich das Modell des Gesamtsystems ergibt. Bei dieser Vorgehensweise können allerdings ungewünschte Abhängigkeiten zu verwendeten Technologien sowie Duplikationen entstehen. Im Bereich Software-Engineering werden verschiedene Strategien zur Extraktion von Modellen aus Quellcode erforscht. Eine Kombination von automatischen Verfahren und manueller Anpassung erzielen die besten Ergebnisse. Im Rahmen dieser Masterarbeit wird ein Ansatz entwickelt, welcher zum Reengineering der Geschäftsprozesse aus Java-Applikationen verwendet wird.

1.2 Aufgabenstellung und Anforderungen

In dieser Masterarbeit wird ein Ansatz zur semiautomatischen Transformation des Quellcodes in Prozessmodelle vorgestellt. Der Benutzer kann manuell auswählen, welche Teile des Systems durch Reengineering verändert werden. So kann man beispielsweise Prozesse aus dem Logik-Modul in Modelle überführen, während Datenbank- bzw. Oberflächenschicht unverändert bleiben. Für den ausgewählten Programmcode werden Modelle generiert. Die erzeugten Modelle bilden den Kontrollfluss des Programms nach. Die Granularität der erstellten Modelle kann vom Benutzer variiert werden. Das Konzept wird als Plugin für das Framework *Java Application Building Center* (kurz: jABC) realisiert. Mit Hilfe des Plugins werden Modelle in Form von *Service Logic Graphen* aus Java-Quellcode erzeugt. Unter anderem sollen folgende Anforderungen bei der Implementierung umgesetzt werden:

- Das Plugin soll eine konfigurierbare Generierung von Modellen ermöglichen
- Der Benutzer soll einstellen können, welche Methoden bzw. Klassen als Modelle dargestellt werden und wie hoch die Granularität der Modellelemente ist.
- Die erstellten Modelle sollen hierarchisch strukturiert werden. Gleiche Quellcode-Abschnitte sollen auf gleiche Untermodelle abgebildet werden.
- Die generierten Modelle sollen syntaktisch korrekt und ausführbar sein. Sie sollen im aktuellen Projekt gespeichert werden und direkt nach der Modellbildung verfügbar sein.
- Bei der Modell-Generierung soll die Semantik des abgebildeten Programmcodes nicht geändert werden.
- Bedienung des Plugins soll über eine grafische Oberfläche erfolgen, die in den jABC-Editor integriert wird.
- Für die Implementierung stehen die Schnittstellen des jABC-Frameworks zur Verfügung. Es muss jedoch berücksichtigt werden, dass diese ggf. weiterentwickelt bzw. modifiziert werden.

Der vorgestellte Ansatz zum modellgetriebenen Reengineering wird im Rahmen der Masterarbeit prototypisch umgesetzt und an einer existierenden Enterprise-Applikation QUMA ausprobiert. QUMA ist ein webbasiertes Qualitätsmanagement-System des Schienenpersonennahverkehrs, das seit 2005 in Nordrhein-Westfalen und seit 2008 in Rheinland-Pfalz verwendet wird. Den Großteil der Anwendung bilden die für Berichterzeugung zuständigen Prozesse. Diese Prozesse werden im Rahmen des Reengineerings in Modelle umgewandelt und modernisiert. Durch die Modellbildung soll die Übersichtlichkeit und Wartbarkeit der Prozesse verbessert werden. Die erzeugten Modelle sollen dem Dokumentationszweck dienen können. Außerdem sollen Erweiterungen und Anpassungen in der Zukunft durch Einsatz von Modellen erleichtert werden. Im Rahmen der Evaluation wird die Qualität der erstellten Modelle untersucht. Als Ergebnis der Arbeit soll bestimmt werden, ob die generierten Modelle zum Reengineering eines echten Systems geeignet sind. Zu berücksichtigen sind folgende Faktoren:

- Übersichtlichkeit und Verständlichkeit der Modelle
- Änderbarkeit und Wartbarkeit des neuen Systems
- Erweiterbarkeit der Prozesse
- Aufwand der Vorbereitungsarbeiten
- Abstraktionsniveau der Modelle
- Benutzerfreundlichkeit der Modelle

1.3 Gliederung der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt. Kapitel 2 führt allgemein in die grundlegenden Themen ein. Der Fokus liegt dabei auf den Begrifflichkeiten im Bereich Reengineering und modellgetriebene Software-Entwicklung sowie auf der Einführung in das

zu verwendete Framework jABC. Im Kapitel 3 wird der zu entwickelte Ansatz zur Modellbildung beschrieben. Nach der Vorstellung des Konzepts wird auf die Umsetzung des Prototyps eingegangen. Der Ablauf und die Ergebnisse des Reengineerings von QUMA werden in Kapitel 4 dargestellt. Hier wird unter anderem mit den zu modernisierenden Geschäftsprozessen auseinandergesetzt. Außerdem werden einige Modelle vorgezeigt, die durch Verwendung des Verfahrens entstanden sind. Verwandte Arbeiten werden im Kapitel 5 diskutiert. Schließlich gibt Kapitel 6 einen Ausblick über offene Forschungsfragen und die mögliche Erweiterungen des Ansatzes.

2 Grundlagen

In diesem Kapitel wird auf die Definition und Probleme von Legacy-Systemen eingegangen. Außerdem werden die Begriffe Reengineering und modellgetriebene Softwareentwicklung erläutert. Als Beispiel für ein Modellierungstool wird das Java Application Buildung Center (kurz: jABC) vorgestellt. Anschließend wird das modellgetriebene Vorgehen beim Reengineering diskutiert.

2.1 Legacy-Software

Viele der praktisch eingesetzten Software-Systeme sind bereits mehr als ein Jahrzehnt alt und bedürfen einer intensiven Wartung und Pflege. Der Bestand an Altsoftware, auch *Legacy-Software* genannt, wird auf über 250 Milliarden Zeilen in alten Programmiersprachen wie Cobol, Fortran oder Assembler, eingeschätzt [53, Kapitel 9].

2.1.1 Definitionen

Zu Legacy-Sytemen gehören nicht nur die Systeme, die in einer veralteten Sprache entwickelt worden sind. Legacy-Software ist eine „geschäftskritische Software, welche nicht, oder nur sehr schwer, modifiziert werden kann“ [35, S. 2]. Auch Softwaresysteme, deren Architektur „sich von modernen Systemarchitekturen deutlich unterscheidet“, gehören zu Legacy-Systemen [10, S. 24]. Einige Wissenschaftler fassen den Begriff deutlich weiter und gehen davon aus, dass heute nicht nur alte monolithische, sondern auch bereits objektorientierte Anwendungen zu den Legacy-Systemen zählen. Der Grund dafür ist die Tatsache, dass objektorientierte Systeme mit wachsender Größe und Komplexität auch schwer zu warten sind:

> „... the empirical evidence is proving that OO is creating new evolution problems and must be used with care to ensure that the complexity of the maintenance is not greater than the complexity of traditional systems.“ [47, S. 76].

Da Legacy-Systeme häufig geschäftskritische Prozesse abbilden, werden diese ungern ausgetauscht und haben dadurch eine besonders lange Lebensdauer. Denn ein Versagen der neuen Software kann ernste Auswirkungen auf tägliche Abläufe im Unternehmen haben. Mit der Zeit wird die Software viel komplexer als ursprünglich konzipiert und kann noch schwerer modifiziert werden. „Es ist schwierig, diese alten Systeme mit den neuen zu verbinden, es ist aufwendig, sie fortzuschreiben, und es ist mühsam, sie zu korrigieren. Es ist auch nicht einfach, Personal zu finden, das sich damit befassen will“ [51, S. 3]. So wird behauptet, dass Software-Systeme, die älter als fünf Jahre sind, bereits zu den Legacy-Systemen gehören [51, S. 3].

2.1.2 Das Entstehen eines Legacy-Systems

Das Entstehen eines Legacy-Systems kann verschiedene Ursachen haben. Einige äußere Einflüsse, wie Entscheidungen des Managements, können dazu beitragen, dass die Software-Systeme nicht modernisiert werden. Neben dem können folgende Ursachen zum Entstehen eines Legacy-Systems führen:

- Das Programm wurde in einer überholten Programmiersprache geschrieben.
- Das System basiert auf einer veralteten Technologien.
- Das System wurde für eine Plattform konzipiert die nicht mehr aktuell ist.
- Bei der Anpassung bzw. Erweiterung des Systems wurden sogenannte „Quick-Fixes“ verwendet, die eine schnelle Lösung eines Problems anbieten, aber die langfristigen Konsequenzen nicht berücksichtigen.

2.1.3 Probleme der Legacy-Software

Veraltete Technologien Legacy-Systeme sind häufig in veralteten Programmiersprachen implementiert oder basieren auf veraltete Betriebs- und Entwicklungsumgebungen. Da immer weniger Entwickler mit den älteren Technologien vertraut sind, sind Legacy-Systeme nur sehr schwer wartbar.

Hohe Komplexität Altsysteme weisen oft eine hohe Komplexität auf. Komplexe Software enthält zahlreiche Schnittstellen, Duplikationen, lange Klassen sowie große Vererbungshierarchien.

Unzureichende Dokumentation Viele Legacy-Systeme zeichnen sich durch fehlende, oberflächliche oder veraltete Dokumentation aus. Die einzige Dokumentation des aktuellen Systems ist der unübersichtliche Code.

Mangel an Expertenwissen Kenntnisse über die bestehenden Programme können im Laufe der Zeit verloren gehen. Personelle Schwankungen innerhalb des Teams führen oft dazu, dass die ursprünglichen Entwickler nicht mehr zur Verfügung stehen. Junge Programmierer verfügen dagegen häufig nicht über die notwendigen Kenntnisse und können die Fachlogik nur ansatzweise oder überhaupt nicht verstehen.

Niedrige Änderbarkeit Die sehr lange Koexistenz von Legacy-Software und Geschäftsprozesse kann zu einer Verknüpfung beider führen [35, S. 3]. Einerseits “lernen” die Geschäftsprozesse die Stärken der Software auszunutzen und die Schwächen zu umgehen. Andererseits sind viele Abläufe, Regeln und Ausnahmen im Legacy-Code “versteckt”, sodass eine Änderung mit hoher Wahrscheinlichkeit zu fehlerhaftem Verhalten des Systems führt.

Hohe Kosten für Wartung und Weiterentwicklung Als direkte Folge der obigen Problemklassen entstehen hohe Kosten für Wartung- und Weiterentwicklungsarbeiten.

2.1.4 Umgang mit Legacy-Code

Es existieren mehrere Strategien, wie man mit dem Altsystem umgehen kann. Welche der Strategien die richtige ist, hängt vom Unternehmen und dem System selbst ab.

Beibehaltung des Altsystem

Obwohl der Einsatz bestehender Legacy-Systeme mit beträchtlichen Problemen verbunden ist, sind viele Unternehmen nicht dazu bereit, die notwendigen Hürden einer Anpassung zu überwinden [50]. Dies ist vor allem dann der Fall, wenn diese Systeme immer noch erfolgreich zur Aufrechterhaltung des laufenden Geschäftsbetriebes beitragen. Leider werden dabei die Probleme des Legacy-Systems nicht gelöst.

Neuentwicklung

Eine Neuentwicklung basierend auf neuen Technologien ist in der Regel zeitaufwendig, kostenintensiv und risikoreich in der Einführung. Die Kosten solcher Entwicklungsprojekte sind in der Regel schwer einzuschätzen. Ein signifikantes Risiko besteht vor allem dann, wenn im System ein Knowhow integriert wurde, welches nirgendwo dokumentiert ist. Bei einer Neuentwicklung kann dieses Wissen verloren gehen.

Ein weiteres Risiko besteht dann, wenn das neu zu entwickelnde System mit anderen Systemen und Ressourcen interagiert. Sind diese Interaktionen nicht klar dokumentiert und verstanden, kann ein Fehler in der Neuentwicklung zu einem Fehlverhalten in den abhängigen Systemen führen.

Ablösung durch Standard-Softwarekomponenten

Legacy-Systeme können durch Integration fertiger kommerzieller Komponenten, wie beispielsweise ERP-Systeme, abgelöst werden. Der Einsatz von Standard-Lösungen ist nur dann empfehlenswert, wenn alle Geschäftsprozesse des Unternehmens mit geringer Anpassung abdeckt werden können. Sonst ist die Ablösung mit hohem Anpassungsaufwand und hohen Einführungskosten verbunden [49, Abschnitt 3.1].

Migration

Unter Migration wird hier eine Überführung eines Softwareprodukts in eine andere technische Umgebung verstanden. Argumente, die für eine Migration sprechen, sind die Erhaltung des integrierten Knowhows sowie der Schutz der im Laufe der Jahre in die bestehenden Systeme geflossenen Investitionen [50]. Im Gegensatz zur Neuentwicklung ist bei Migration eine zuverlässigere Aufwandsschätzung möglich, denn die Größe des Systems (und des Codes) ist bekannt.

Es existieren folgende Möglichkeiten, eine Migration durchzuführen [51, Kapitel 5]:

- Kapselung (engl. wrapping),
- automatische Konvertierung und
- Reengineering

Bei der *Kapselung* bleiben Daten und Programmteile des Altsystems in ihrer ursprünglichen Umgebung (alte Plattform und alte Programmiersprache). Sie werden von einem sogenannten *Wrapper* umhüllt, der das Altsystem kapselt und entsprechende Zugriffsschnittstellen implementiert, über die das Neusystem auf die Komponenten des Altsystems zugreifen kann.

Eine automatische *Konvertierung* von Programmen des Basissystems in Programme des Zielsystems ist nur für bestimmte Teile des Systems möglich. Beispielsweise kann der Programmcode mit Hilfe sogenannter Translatoren von einer Programmiersprache in eine andere automatisch übersetzt werden. Eine Datenbank kann mithilfe bestimmter Tools von einem Dialekt in einen anderen automatisch überführt werden.

Beim *Reengineering* wird das Altsystem in neue Strukturen transformiert ohne das Verhalten des Systems zu ändern. Die Programmiersprache bzw. die verwendeten Technologien können erhalten bleiben. Durch Veränderungen im Design oder in der Architektur kann durch Reengineering die Übersichtlichkeit und Wartbarkeit des Legacy-Systems verbessert werden. Da automatische Migration oft unmöglich ist und das Wrapping die Probleme der Legacy-Software nicht löst (sondern nur versteckt), ist das Reengineering die am meisten verwendete Migrationsstrategie [35, Abschnitt 5.6].

2.2 Reengineering

2.2.1 Definitionen

Der Begriff *Software-Reengineering* wird in der Literatur sehr uneinheitlich verwendet. Eine eindeutige Definition gibt es nicht, deshalb werden im Folgenden mehrere gängige Definitionen vorgestellt. Alle Definitionen enthalten die Einschränkung, dass trotz aller vorgenommenen Veränderungen die fachliche Funktionalität unangetastet bleibt. Wenn beim Reengineering neue Funktionalitäten hinzugefügt oder die alten verändert werden, spricht man von *erweiterndem Reengineering* [4].

Software-Reengineering wird im Ansatz von Chikofsky und Cross [9] basierend auf drei Phase des Software-Entwicklungsprozesses (Anforderungen, Entwurf und Implementierung) ausführlich behandelt. Abbildung 2.1 zeigt das Begriffsmodell der Reengineering-Taxonomy. Chikofsky und Cross definieren Reengineering als „Untersuchung und Modifikation eines Systems, um es in einer neuen Form wiederherzustellen und diese Form nachfolgend zu implementieren“ [4, S. 193].

> „... the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form“ [9, S. 14]

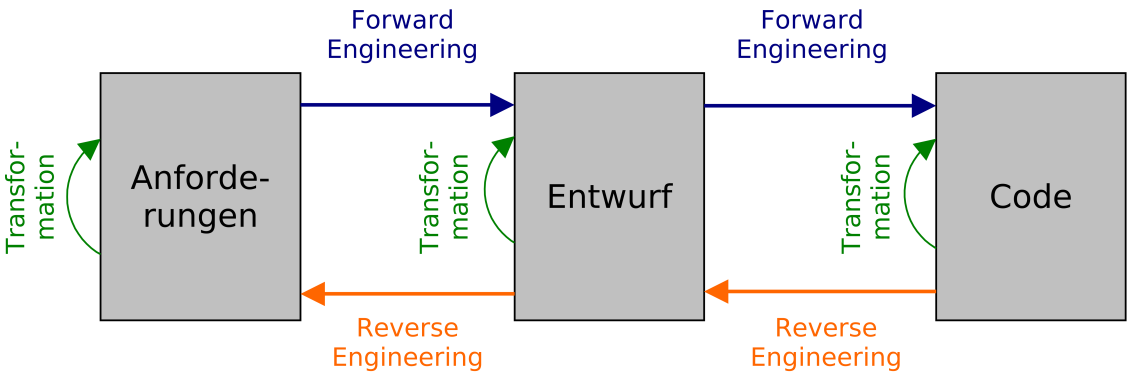


Abbildung 2.1: Reengineering laut Chikofsky und Cross [9]

Die Wissenschaftler unterteilen den Prozess des Reengineerings in drei Schritte. Im ersten Schritt wird das System auf höherem Abstraktionsniveau beschrieben, dieser Schritt wird als *Reverse Engineering* bezeichnet. Durch Reverse-Engineering werden Informationen aus Software-Artefakten extrahiert mit dem Ziel Strukturen, Verhaltensweisen und Zusammenhänge von Elementen bestehender Software-Systeme zu untersuchen. Darauf aufbauend wird das System durch eine *Transformation* verändert und abschließend im Rahmen des *Forward-Engineerings* neu implementiert. Forward-Engineering entspricht dem traditionellen Software-Entwicklungsprozess. Es beschreibt die stufenweise Verfeinerung von einem höheren hin zu einem niedrigeren Abstraktionsniveau, konkret von der Anforderung hin zum Quellcode.

Während Chikofsky und Cross ihre Definition auf dem Prozess der Softwareenwicklung basieren, untersuchte Byrne [7] die verschiedenen Ebenen eines Softwaresystems, von der Ebene des Quellcodes bis zur Systemarchitektur. Reengineering wird dabei als Folge der drei Schritte *Abstraktion*, *Änderung*, *Verfeinerung* verstanden. Das zentrale Konzept des Reengineerings stellt die Abstraktion dar. Unter Abstraktion versteht man hier den Übergang in eine höhere Repräsentationsebene. Zum Beispiel werden die For-, Foreach- oder While-Schleifen auf der Ebene des Quelltextes als Abfolge von Iterationsschritten auf der Ebene der Code-Struktur repräsentiert.

Die genauen und detailliertesten Informationen befinden sich in der untersten, also der Implementierungsebene. Die abstraktesten Repräsentationen sind auf der Design-Ebene zu finden. Änderungen innerhalb einer Ebene haben ausschließlich Einfluss auf die darunter liegenden, jedoch nicht auf die darüber liegenden Ebenen. Wird eine Foreach-Schleife durch eine For-Schleife ersetzt, ändert sich die äußere Code-Struktur nicht. Wenn aber eine Anforderung geändert werden muss, kann die Architektur betroffen und einzelne Teile des Systems müssen eventuell neu entworfen und implementiert werden.

Basierend auf dem Modell von Byrne wurde an der Carnegie Mellon Universität das sogenannte „Horseshoe“-Modell [28] entwickelt. Dieses wird in Abbildung 2.2 dargestellt. Das Modell teilt die verschiedenen Ebenen in Quellcode, Funktion und Architektur ein. Die drei Schritte des Reengineerings (Rekonstruktion, Transformation und Verfeinerung) können auf allen Ebenen stattfinden.

McClure [37] erweitert die Definition des Reengineerings um Untersuchung und Modifikation von Software mit Hilfe automatisierter Werkzeuge. Diese werden genutzt, um Aspekte wie Wartbarkeit und Lebensdauer zu verbessern, Technologien zu ersetzen und die Produktivität der Wartung zu steigern. Arnold [2] behauptet, dass Reengineering nicht vollständig automatisierbar ist. Ein automatisierter Test der semantischen Äquivalenz des alten und neuen Systems sei nicht möglich. Laut Baumöl et al [4] wird Software-Reengineering als das Forschungsgebiet bezeichnet, „das sich mit der Entwicklung von Methoden, Techniken und Vorgehensweisen beschäftigt, die eine Produktivitätssteigerung und Qualitätsverbesserung ermöglichen“. Ähnlich bezeichnet Ebert Reengineering als ein „Oberbegriff für alle Prozesse, deren Ziel die qualitative Verbesserung und Aufbereitung von Software ist“ [10].

Reengineering und Refactoring

Viele Begriffserklärungen des Reengineerings basieren auf dem Konzept Abstraktion und beschreiben eine Erhöhung des Abstraktionsniveaus im ersten Schritt. Wenn beim Reen-

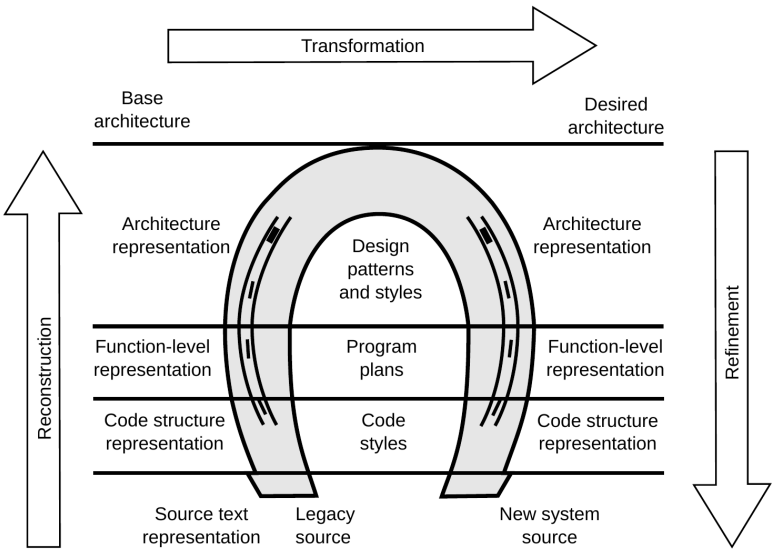


Abbildung 2.2: Das Horseshoe-Modell [28]

gineering das Abstraktionsniveau unverändert bleibt, spricht man von *Restrukturierung*. Unter Restrukturierung versteht man die Transformation von einer Repräsentationsform in eine andere ohne Änderung der Funktionalität. Restrukturierung findet immer innerhalb einer Abstraktionsebene statt:

> „Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)“ [9, S. 14].

Auf der Code-Ebene wird Restrukturierung häufig als *Refactoring* bezeichnet:

> „Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure“ [19, S. xvi].

Zu den Refactoring-Aufgaben gehören zum Beispiel die Einführung eines einheitlichen Programmierstils (Coding Standards), das Verbessern des Kontrollflusses, das Entfernen von duplizierten Code-Stellen und andere.

2.2.2 Ziele und Techniken

Reengineering hat unter anderem die Ziele, das Verständnis der Software zu erneuern, die Software zu erweitern, die Wart-, die Wiederverwendbarkeit und die Erweiterbarkeit zu verbessern [2, Kapitel 1]. Weitere Ziele, die beim Reengineering verfolgt werden können, werden im Folgenden aufgelistet.

- Kontrolle der Komplexität
- Wiedergewinnung verlorenen Informationen
- Unterstützung von Wiederverwendung
- Gewinnung alternativer Sichten

- Erkennung von Seiteneffekten
- Schaffung höherer Abstraktionen

Für die Lösung von Reengineering-Aufgaben, vor allem von Reverse-Engineering-Aufgaben, können folgende Basistechniken verwendet werden [43]:

- *Parsing*, d.h. die Überführung von Programm in abstrakte Darstellungen
- *Kontrollfluss-Bestimmung*, d.h. die Berechnung der möglichen Verzweigungen bei der Ausführung von Programmen
- *Datenfluss-Berechnung*, d.h. die Ermittlung der Beziehungen zwischen Zuweisungen an Variablen und deren Benutzung
- Berechnung des *Programm-* bzw. *Klassen-Abhängigkeitsgraphen*

Für den Umgang mit den berechneten Informationen verwendet man oft noch weitere Techniken zur Strukturierung, wie beispielsweise

- **Konzeptanalyse** zur Strukturierung von Informationen anhand ihrer Attribute,
- **Clusteranalyse** zur Gruppierung von Informationen aufgrund ihrer Ähnlichkeit,
- **Visualisierung** zur Darstellung von ermittelter Information in für den Menschen adäquater Form

2.2.3 Reengineering-Projekte

Reengineering-Projekte können an verschiedenen Ebenen von Softwaresystemen ansetzen. Im Rahmen des Reengineerings können zum Beispiel reine Sprachmigrationen (Änderung der Programmiersprache) durchgeführt werden, es können Architekturmigrationen (monolithische zu service-orientierter Architektur, Mainframe hin zu Client-Server-Architektur) in Angriff genommen werden oder es kann auf der Ebene der Geschäftsprozesse angesetzt werden (engl. Business Process Reengineering). Außerdem kann Reengineering eine einfache Datenbankumstellung sowie bei der Migration auf andere Hardware oder auf ein anderes Betriebssystem umfassen. Die Umstellung eines Kommandozeilen-Tools auf ein System mit grafischen Benutzeroberfläche ist ein weiteres Beispiel eines Reengineering-Projektes. Kombinationen der Arten sind ebenfalls denkbar.

2.2.4 Werkzeuge

Reengineering-Werkzeuge unterstützen den Zyklus Abstraktion-Transformation-Implementierung. Sie bieten mehrere Basistechniken und deren Kombination an, damit verschiedene Reengineering-Aufgaben gelöst werden können. Zusätzlich ist eine Integration in Programm-Entwicklungs-Werkzeuge für das Forward Engineering möglich. Es existieren zahlreiche Werkzeuge, die verschiedene Technologien verwenden. Viele Reengineering-Tools sind Repository-basiert, sie arbeiten auf einer zentralen Datenhaltung (Repository). Die Spannbreite der angewandten Formate für die Datanhaltung reicht von Text-Dateien über relationale Datenbanken und Graphenspeicher bis hin zu Prolog-Klauselmengen und XML-Files. Im Folgenden werden drei Reengineering-Tools vorgestellt.

An der Victoria Universität in Kanada wurde ein Reverse-Engineering-Werkzeug **Rigi** entwickelt [30]. Rigi automatisiert Aktivitäten im Reverse-Engineering, um die Verarbeitung von großen Programmen zu ermöglichen. Laut den Autoren war Rigi eines der ersten Werkzeuge, welches Softwareartefakte sowie deren Beziehungen untereinander graphisch darstellt und deren interaktive Manipulation ermöglicht.

Das **Columbus**-Reengineering-Werkzeug [18] ist im Rahmen einer Kooperation der Universtät Szeged sowie der Unternehmen Nokia und FrontEndArt entstanden, um verschiedene Aufgaben des Software-Reengineerings zusammenzufassen. Columbus unterstützt das Reengineering von in C++ entwickelter Software. Dazu besitzt das Tool einen leistungsfähigen C++-Parser sowie ein Schema für eine abstrakte Repräsentation des Quellcodes.

GUPRO [11] ist eine Umgebung, welche die verschiedenen Abstraktionsebenen eines Software-Systemes in Graphmodellen abbilden kann. Die Basis für die abstrakte Darstellung bilden sogenannte *TGraphen*. Aus einem existierenden System wird zunächst eine Graph-Datenbank erstellt. Mithilfe einer Abfragesprache können aus dieser Datenbank Informationen über das System gewonnen und Meta-Modelle erstellt werden.

2.3 Modellgetriebene Softwareentwicklung

Die *modellgetriebene Softwareentwicklung* (engl. Model-Driven Software Development, kurz MDSD) befasst sich mit der Automatisierung in der Softwareherstellung. Modellgetriebene Softwareentwicklung ist ein Oberbegriff für "Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen" [54, S. 12]. In der modellgetriebenen Entwicklung werden möglichst viele Artefakte eines Softwaresystems automatisch generiert. Bei den Artefakten handelt es sich hauptsächlich um den Quellcode und andere Dateien, die für die Lauffähigkeit des Systems benötigt werden, z.B. Konfigurationsdateien oder Datenbankskripte. Darüber hinaus können auch entwicklungsunterstützende Artefakte generiert werden, z.B. Softwaretests und Dokumentation [42, Abschnitt 2.1].

Die entscheidende Grundlage für die Generierung der Artefakte des Softwaresystems sind „hinreichend formale und zugleich abstrakte *Modelle*, in denen bewusst Details für eine kompakte Übersicht weggelassen werden und die Architektur oder Funktionalität adäquat beschreiben wird" [S. 12][42]. Modelle beziehen sich auf einen bestimmten Fachbereich, sie orientieren sich am Problemraum der jeweiligen Domäne [54, Abschnitt 3.1]. Zur Formalisierung der Modelle wird eine höhere *domänenspezifische Modellierungssprache* benötigt, die sogenannte DSL. Modelle nehmen bei MDSD die zentrale Stellung, weil sie auch für Beschreibung und Dokumentation des Systems verwendet werden. Sie werden als Bestandteil der Software behandelt und stellen einen ganz entscheidenden Beschleunigungs- und Qualitätsfaktor dar.

2.3.1 Begrifflichkeiten[1]

Domäne

Der Ausgangspunkt der Modellierung ist ein begrenztes Wissensgebiet, oder eine Domäne, die eine Menge von Konzepten umfasst. Beispielsweise kann die Domäne *Versicherungen*

[1] basierend auf [42, Abschnitt 1.3]

durch die Konzepte Tarif, Schaden, Leistung, Vertrag usw. dargestellt werden.

Plattform

Eine Plattform stellt den Lösungsraum zur Umsetzung der Domäne dar. Plattformen bestehen aus Bausteinen, wie z.B. Bibliotheken, Frameworks und Middleware.

Modell

Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten einer Domäne. Semantisch stellt ein Modell einen in einer domänenspezifischen Modellierungssprache formulierten Satz dar. Die Bedeutung eines Modells ist von der Bedeutung einzelner Modellelemente, also Konzepte aus der Domäne, abhängig.

Ein Schlüsselkonzept der modellgetriebenen Softwareentwicklung ist die Trennung von plattform-unabhängigen Modellen (engl. Platform Independet Model, kurz PIM) und plattform-spezifischen Modellen (engl. Platform Specific Model, kurz PSM). Die ersten bilden die Fachlichkeiten ohne technologische Details ab, während die letzten Konzepte einer Plattform verwenden, um ein System zu beschreiben. Wenn beispielsweise ein konkretes System basierend auf der Java-2-Enterprise-Edition (J2EE) [62] zu realisieren ist, ist das PIM die Beschreibung des Systems ohne J2EE-spezifische Details, während das PSM einem mit J2EE-spezifischen Details angereicherten Modell entspricht. In dem PSM werden Entitäten wie z.B. „Tarif“ und „Vertrag“ als JavaBeans dargestellt.

Transformationen

In der MDSD existieren zwei Arten von Transformationen. Bei den *Modell-zu-Modell* Transformationen handelt es sich um Transformationen, die ein Quell-Modells auf ein Ziel-Modell abbilden. Eine Transformation beschreibt die Regeln, wie einzelne Elemente abzubilden sind. Zum Beispiel können aus einem PIM durch Modell-zu-Modell Transformationen PSMs erstellt werden.

Eine andere Art der Transformation ist *Modell-zu-Plattform*. Das Ergebnis dieser Transformation ist nicht ein abstraktes Modell, sondern eine plattform-spezifische Implementierung. Code-Generierung ist ein Beispiel für diese Transformation.

2.3.2 Code-Generierung

Code-Generatoren sind Metaprogramme, also Programme, die andere Programme erzeugen. Code-Generatoren bekommen die Modelle als Eingabeparameter und erzeugen Quelltext als Ausgabe [54]. Als Eingabe werden üblicherweise Modelle in grafischer oder textueller Notationen verwendet (zum Beispiel als XML-Dateien).

Es gibt unterschiedliche Ansätze und Werkzeuge zur Codegenerierung mit jeweils unterschiedlichen Zielsetzungen. Zum einen existieren die „geschlossenen Ansätze, die vorgefertigte - aber durchaus optimierte - Lösungen für ganz spezifische Problemstellungen bieten“ [60, S. 7]. Das andere Extrem sind die „offenen Systeme, die keine Vorgaben über den zu generierenden Code und dessen Struktur machen“ [60, S. 8]. Sie stellen dem Entwickler die grundlegenden Mechanismen für Codegenerierung zur Verfügung. Dazu zählen

Template-basierte Systeme und Frame-Prozessoren genauso wie API-basierte Generatoren.

Template-basierte Generatoren

Die Template-basierten Generatoren verwenden vorgefertigte Vorlagen (engl. templates) für das fertige Programm, die in textueller Form vorliegen [3]. In den Vorlagen sind einige fest vorgegebene Programmteile eingetragen. An manchen Stellen stehen aber nur Platzhalter da. Diese Platzhalter werden durch den Generator unter Berücksichtigung des Kontextes ersetzt, um den Code des Zielprogramms zu erhalten. Der Kontext wird vom Generator aus dem Modell bzw. der Konfigurationsdateien erstellt. Der Vorgang der Zusammenführung eines Templates mit dem Kontext wird als *Merging* bezeichnet. Der große Vorteil der Code-Generierung mit Templates ist die leichte Anpassbarkeit des Generators, da in vielen Fällen nur die Templates geändert werden müssen. Prominente Vertreter sind das Velocity-Projekt [57] und die Transformationssprache XSLT [56].

Frame-Prozessoren

Frame-Prozessoren ist eine Generatortechnik, bei der Quellcode aus einer Spezifikation generiert wird. Frames sind prinzipiell Codebausteine, die sich wie Objekte benutzen lassen. Die Verwendung von Frames ähnelt der objektorientierten Programmierung, bei der zwischen Frames und Frame-Instanzen unterschieden wird. In den Frames sind so genannte Slots untergebracht, bei denen es sich um Variabilitätspunkte handelt, die in verschiedenen Instanzen unterschiedlich ausgeprägt sein können. Ausgeprägte Slots enthalten textuelle Bausteine, wie zum Beispiel Code-Blöcke, oder sie referenzieren wiederum Frame-Instanzen [15]. Die Aufgabe des Entwicklers besteht darin, aus Modellen die Frames zu erstellen, der Frame-Prozessor kann danach aus Frames den Quellcode in einer beliebigen Sprache generieren. Die Vorteile von Frame-Prozessoren umfassen die Wiederverwendbarkeit von Frames und einfache Umstellung auf eine andere Programmiersprache. Ein Frame-Generator wird beispielsweise im Projekt XFramer [15] verwendet.

API-basierte Generatoren

Eine Alternative zur oben beschriebenen Generatoren stellen die API-basierte Generatoren dar. Der gesamte Aufbau des zu generierenden Artefaktes wird über eine abstrakte Schnittstelle beschrieben [58, Abschnitt 6.2.6]. Der Generator „kennt“ nur die Schnittstelle, die durch den Entwickler manuell zu realisieren ist. Der manuell implementierte Teil enthält den technischen Code z.B. für die Speicherung, Anzeige, Fehlerbearbeitung und ähnliche standardisierbare technische Funktionalitäten. Der Generator erzeugt den sogenannten „Anwendungscode“, der den technischen Code per Schnittstelle verwendet. API-basierte Generatoren nutzen das hohe Potential zur Wiederverwendung des technischen Codes und sind sehr flexibel. Ein Beispiel für einen Generator dieser Art stellt das Projekt iText [23] dar.

Inline-Generierung

Sind im regulären Quelltext Konstrukte erhalten, die beim Kompilieren des Codes weiteren Maschinencode erzeugen, spricht man von Inline-Generierung [58, Abschnitt 6.2.6]. Als Beispiel dieser Generierungstechnik dienen Anweisungen an den Präprozessor in C oder C++.

Bootstrapping

Es ist schwierig und manchmal nicht sinnvoll, im Projekt direkt mit der Template- oder Frame-Programmierung zu beginnen, wenn noch keine generative Softwarearchitektur vorliegt. Stattdessen sollte der später generierte Code zunächst von Hand beispielhaft ausgearbeitet werden. Erst danach können Templates bzw. Frames abgeleitet werden. Man kann beispielsweise als Basis eine lauffähige Referenzimplementierung der Applikation nehmen und die statischen Codefragmente eins-zu-eins in die Templates übertragen. Die variablen Teile werden mit Hilfe der Template-Sprache formuliert [54, Abschnitt 8.5].

2.3.3 Werkzeugunterstützung

Bei der Modellierung komplexer Softwaresysteme ist ein Modellierungswerkzeug unverzichtbar. Durch Verwendung von Werkzeugen kann die Produktivität deutlich verbessert werden. Bei der Auswahl von Modellierungstools kommt es vor allem auf Qualität und Komfort der Benutzung an. Ein Tool sollte die Eigenschaften und Spezifikationen von Modellelementen übersichtlich darstellen und einfach verändern lassen. Die Modelle sollen in einem portablem Format (z.B. XML) gespeichert werden. Auch die Anpassbarkeit der Domäne ist wichtig. Wünschenswert wäre eine automatische Anbindung von Benutzerprofilen und die Teamfähigkeit des Tools. Viele Modellierungsframeworks bieten Plugins zur Validierung der Modelle an. Die benutzerdefinierten Validierungsregeln werden während der Modellierung oder beim Speichern des Modells aktiviert. Die Fehler sollten dabei am Modellelement angezeigt werden und nicht in einer separaten log-Datei.

Es existieren viele Werkzeuge zur Unterstützung von Erstellung und Bearbeitung der Modelle. Einige werden im Folgenden vorgestellt. Ein Beispiel eines Eclipse-basierten [13] Tools ist das Eclipse Modeling Framework (kurz EMF) [14] zur automatisierten Erzeugung von Quelltext anhand von strukturierten Modellen. Mithilfe des Graphical Modeling Project (kurz GMP) [21] können eigene Editoren auf Basis von Eclipse erstellt werden. Das im vorherigen Abschnitt erwähnte Framework GUPRO unterstützt bei der Modellierung der TGraph-Modellen. Zu den kommerziellen Lösungen zählt Rational Rhapsody [45], eine UML-basierte grafische Umgebung für Entwicklung der Echtzeit- oder Eingebettete Softwaresysteme. An der Technischen Universität Dortmund und an der Universität Potsdam wurde das Java Application Building Center (kurz jABC) [55] entwickelt. Dieses Framework wird im Rahmen dieser Masterarbeit verwendet und im Abschnitt 2.5 detailliert vorgestellt.

2.3.4 Gründe für modellgetriebene Entwicklung

Die Gründe, modellgetriebene Softwareentwicklung zu verwenden, können vielfältig sein. Die wichtigsten werden im Folgenden beleuchtet.

- **Abstraktion und Wiederverwendbarkeit** sind die wichtigsten Gründe für MDSD, da mit Modellen auf einer höheren Ebene programmiert werden kann [54]. Die Modelle beschreiben das System mit größeren Bausteinen, die wiederverwendbar sind.

- **Steigerung der Effizienz und Produktivität, Reduzierung von Zeit und Aufwand der Entwicklung**, da aus formalen Modellen lauffähiger Code generiert wird und Modellelemente wiederverwendbar sind. Modelländerungen werden

schnell in Code umgesetzt. Domänenspezifische Modelle sind für Fachexperten besser verständlich, da diese von technischen Details befreit sind. Somit können fachliche Änderungen direkt auf Modellen umgesetzt werden.

- **Steigerung der Software-Qualität** durch wohl definierte, einheitliche, verbindliche Architektur und Trennung von Fachlichkeit und Technik. Modelle, Anwendung und Dokumentation sind stets konsistent, denn die gesamte Implementierung inklusive Dokumentation ist in Modellen enthalten und der Quellcode wird zum größten Teil generiert. Die Qualität von generiertem Quelltext bleibt erhalten. Die Anzahl der Fehler im Code kann reduziert werden, da der Code nicht per Hand geschrieben wird und im Generator vorhandene Fehler müssen nur einmal beseitigt werden.

- **Trennung von Verantwortlichkeiten** durch Aufteilung in die beiden Rollen Fachanwendungsentwickler und Framework- bzw. Generator-Entwickler. Die Trennung der Implementierung und Modellierung ermöglicht die Konzentration der jeweiligen Entwickler fachliche bzw. technische Aspekte. Die Arbeit kann besser verteilt und jedem Entwickler kann ein Spezialgebiet zugewiesen werden.

- **Bessere Wartbarkeit, Standardisierung und Portabilität** werden erreicht, da fachliche Änderungen auf der Ebene der Modelle durchgeführt werden. Beim Wechsel der technischen Plattform muss lediglich der Generator angepasst werden.

Ob sich die aufgeführten Vorteile der MDSD in der Praxis umsetzen lassen, hängt von verschiedenen Voraussetzungen ab. Der Einsatz von MDSD benötigt initiale Investitionen für die Erst-Erstellung des Generators und ein geeignetes Modellierungsframework. Investitionen sind insbesondere nötig, um die Mitarbeiter auszubilden, einen geeigneten Generator zu entwickeln und das Vorgehen anzupassen. Außerdem müssen organisatorische Rahmenbedingungen vor allem durch Umstellung von Vorgehensmodellen und Projektmanagement geschaffen werden [42].

2.3.5 Best Practices[2]

Der erfolgreiche Einsatz der modellgetriebenen Softwareentwicklung erfordert einige Praktiken, die in verschiedene Bereiche unterteilt werden können.

Code-Generierung

Der generierte Code soll genau so wie der handgeschriebene möglichst gut aussehen. Es ist unrealistisch anzunehmen, dass Entwickler den generierten Code nie zu sehen bekommen. Selbst wenn die Anwendungsentwickler den Code nicht ändern müssen, werden diese mit dem generierten Code konfrontiert, wenn sie zum Beispiel die Anwendung debuggen müssen. Daher sollte man Kommentare mitgenerieren lassen und Tools für Code-Formatierung verwenden [42].

Um die Komplexität des Generators zu verringen, können die Modelle zunächst mit Hilfe der Modell-zu-Modell-Tranformationen in eine Zwischenpräsentation überführt werden. Erst danach kann der Generator angewandt werden.

[2] basierend auf [42, Kapitel 5]

Der generierte Code soll vom nicht-generierten Code getrennt werden. Eine Modifikation des generierten Codes kann Probleme in Sachen Konsistenz, Versionierung und Build-Management verursachen. Bei Trennung zwischen generierten und manuell erstelltem Code (in verschiedenen Dateien und Verzeichnissen), sollte der generierte Code nicht modifiziert werden. Der generierte Code kann als Wegwerfprodukt angesehen werden, welches noch nicht einmal versioniert werden muss. Eine Integration des generierten Codes kann dann mithilfe von Interfaces, abstrakten Klassen oder die Entwurfsmuster *Factory*, *Strategy*, *Bridge* und *Template* [17] geschehen. Die Architektur soll klar definieren, welche Artefakte generiert und welche manuell erstellt werden. Wenn der generierte Code aus irgendeinem Grund doch manuell modifiziert werden muss, dann ist die Einführung von speziellen geschützten Bereichen unvermeidbar. Diese Bereiche sollen im Generator berücksichtigt und bei der Neugenerierung nicht überschrieben werden.

Versionierung

Bezüglich Versionierung sind Modelle und einzelne Modellelemente genauso zu behandeln wie Quelltext. Verteilte Teams müssen also in der Lage sein, Modelle aus einem Versionsverwaltungssystem auszuchecken, einzelne Modellelemente zu bearbeiten, mit früheren Versionen zu vergleichen, konkurrierende Modifikationen zu bearbeiten und den neuen Stand wieder einzuchecken.

Tests

Ähnlich wie bei der manuellen Softwareentwicklung spielen Tests auch bei MDSD eine große Rolle. Modelle sind Teile des Software-Systems, in denen Funktionalitäten implementiert sind. Modelle sollen genau so wie manuell entwickelter Code getestet werden. Auch der testgetriebene Ansatz ist mit der modellgetriebenen Entwicklung kombinierbar. Die Entwickler können vor der Modellierung Tests schreiben. Bei der modellgetriebenen Entwicklung können die Testszenarien (Spezifikationen von Ein- und Rückgabewerten) für Blackbox-Tests mithilfe der Modelle erstellt werden. Es kann effizienter und verständlicher sein, die Testdaten (Vor- und Nachbedingungen eines Testfalls) zunächst im Modell zu definieren. Aus den Testdaten können die Testfälle generiert werden. Somit erreicht man die Trennung zwischen Daten und eigentlichen Tests.

Evolution der Modelle

Bei der modellgetriebenen Entwicklung soll berücksichtigt werden, dass im weiteren Verlauf die erstellten Modelle oft modifiziert werden müssen. Eine Änderungen in der domänenspezifischen Sprache der Modelle kann Fehler in existierenden Modelle verursachen. Durch flexible Struktur oder Einstufung von Sprachelementen als *deprecated* sind mögliche Lösungen.

2.4 Extreme Model-Driven Design

Das extreme Model-Driven Design(kurz: XMDD) [32, 33] ist ein serviceorientierter Entwicklungsansatz, in welchem die klassische modellgetriebene Entwicklung mit Prinzipien des *Extreme Programming* [5] kombiniert wird. Wie im klassischen MDSD sind Modelle die zentralen Artefakte des XMDD. Sie repräsentieren eine Abstraktion des Gesamtsystems und dienen als Vorlage für die Generierung des Programmcodes. Im Unterschied zu

MDSD, in welchem Anwendungsexperten bei der Modellierung oft nur als Berater beschäftigt werden, sieht XMDD die Integration der Anwendungsexperten in den gesamten Ablauf einer Produktentwicklung vor. Außerdem benutzt XMDD das sogenannte *One-Thing-Approach*, welches die Repräsentation aller Aspekte des zu entwickelnden Systems in einem Modell ermöglicht. Durch die Service-Orientierung wird die Ausführbarkeit der Modelle gewährleistet.

2.4.1 One-Thing-Approach

Im klassischen modellgetriebenen Vorgehen sind verschiedene Arten von Modellen erlaubt, wie zum Beispiel Klassen-, Aktivitäts- bzw. Anwendungsfalldiagramme in UML [41]. Das extreme Model Driven Design benutzt das sogenannte One-Thing-Approach, welches eine einzige Modellvariante für den gesamten Lebenszyklus einer Software definiert. Dieses Modell beinhaltet alle Facetten des zu beschreibenden Systems, welche in Abbildung 2.3 veranschaulicht werden:

- Anwendungs- bzw Geschäftsprozesse
- Datenstruktur
- Software Architektur
- Implementierung
- Dokumentation

Das One-Thing-Approach setzt einen vollständig automatischen Codegenerierungsprozess voraus. Alle Änderungen werden ausschließlich an den Modellen vorgenommen, der generierte Code wird nicht manuell verändert.

2.4.2 Service-Orientierung

XMDD verwendet Technologie-unabhängige Prozessmodelle, die ausführbar und verifizierbar sind. Dazu werden diese, zusammen mit den Anwendungsexperten, solange verfeinert, bis atomare Funktionalitäten auf unterster Ebene entstehen. Diese nicht weiter zerlegbaren Teile werden von *Services* realisiert. Unter einem Service wird hier ein eigenständiges,

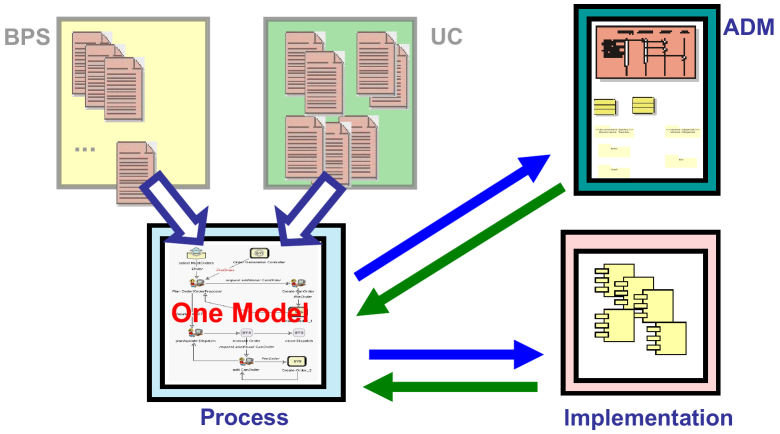


Abbildung 2.3: One Thing Approach(vgl. [33])

plattformunabhängiges Softwaremodul verstanden, welches eine öffentliche Schnittstelle definiert und dynamisch in das Prozessmodell eingebunden werden kann. Services können ein breites Spektrum an Funktionalitäten abdecken: von klassischen Methoden bis hin zu verteilten Technologien wie Middleware-Systeme oder Web-Services [32].

XMDD-Modelle stellen universelle Lösungen in der spezifischen Anwendungsdomäne dar und können durch Anbindung von Services in verschiedenen Technologien realisiert werden. Das bedeutet, dass ein Technologie-Wechsel ohne Anpassung des Modells und nur durch Austauschen von Service-Implementierungen erfolgen kann. Außerdem bietet die Service-Orientierung eine Grundlage zur Wiederverwendbarkeit von gleichen Services in verschiedenen Modellen.

2.4.3 Integration der Anwendungsexperten

XMDD ermächtigt den Anwendungsexperten schon von der frühen Entwicklung eines Produkts an, nicht nur als Beobachter und Berater aufzutreten, sondern direkt in die Entwicklung einzugreifen [33]. Bei der Modellierung benötigt der Anwendungsexperte keine Kenntnisse der Implementierungstechnologien, denn die in Modellen verwendeten Services plattformunabhängig sind. Die existierenden Services können in Modellen wiederverwendet werden. Noch zu entwickelnden Funktionalitäten werden an die Entwickler weitergegeben und von diesen implementiert. Ähnlich wie Extreme Programming unterstützt XMDD die gemeinsame kontinuierliche Weiterentwicklung und Verbesserung der Geschäftsprozesse. Auf Modellebene entstehen während der Entwicklung verschiedene Prototypen, die jederzeit zu Review, Test sowie Simulation zur Verfügung stehen. Anwendungsexperten können so den Fortschritt besser kontrollieren und auf die Gesamtentwicklung direkt Einfluss nehmen.

2.5 Java Application Building Center

Bei dem Java Application Building Center (kurz jABC) handelt es sich um ein modulares Framework zur Modellierung von komplexen Software-Systemen. jABC wird an der Technischen Universität Dortmund sowie der Universität Potsdam entwickelt. Ein Vorläufer des Programms, das Application Building Center (kurz ABC), entstand bereits Mitte der 90er Jahre und war in C++ implementiert [61]. Die aktuelle Version 4 des jABC ist in Java umgesetzt und besitzt eine Schnittstelle zur direkten Ausführung von Java-Code [55, 40].

Die Modellierung von Prozessen im jABC basiert im Wesentlichen auf dem Konzept des extremen Model Driven Development. Geschäftsprozesse können mit Hilfe des jABC als hierarchisch strukturierte *Service Logic Graphen* (kurz: SLG) bestehend aus funktionalen Einheiten, den sogenannten *Service Independent Building Blocks* (kurz: SIB), modelliert werden [55]. Ein SIB wird als Knoten im Prozessgraph dargestellt. Die Verbindung zwischen SIBs wird durch gerichtete Kanten, sogenannte *Branches*, realisiert. Bereits erstellte Modelle können innerhalb anderer Modelle verwendet werden, indem sie durch einen *GraphSIB* referenziert werden. Dadurch ist eine hierarchische Strukturierung der Modelle möglich. Eine Besonderheit von jABC ist die Möglichkeit, modellierte Prozesse direkt auszuführen. Dies wird ermöglicht durch die Verwendung der Programmiersprache Java, die plattformunabhängig ist. Die Modelle könne als XML-Dokumente gespeichert und versioniert werden. ermöglicht eine einfache Versionierung.

Mit jABC können die fachliche Definition der Prozesse und die Implementierung der Funktionalität getrennt werden. Das Prozessdesign kann von den *Fachexperten* durchgeführt werden, auch wenn diese keine Informatik-Kenntnisse besitzen. Denn zum Modellieren verwendet der Fachexperte die durch den *SIB-Experten* bereitgestellten SIBs als Bausteine. Die Aufgabe der SIB-Experten besteht aus der technischen Implementierung einzelner SIBs.

2.5.1 Modellierung

Modelle werden in jABC auch Service Logic Graphen genannt. Ein SLG hat genau einen Startknoten und deren Kanten (auch Branches genannt) sind gerichtet und beschriftet. Die Namen der Branches repräsentieren das Ergebnis der Ausführung des Startknotens. Untergeordnete SLGs lassen sich in obergeordneten referenzieren, wodurch eine Hierarchie von Modellgraphen entsteht. In Abbildung 2.4 ist eine solche Hierarchie dargestellt.

Die vorgefertigten Bauelemente, die funktionale Einheiten für die Prozessmodelle darstellen, werden als Service Independent Building Blocks bezeichnet. Ein SIB ist eine Java-Klasse, die das Interface `SIB` implementiert und eingegrenzte Funktionalität ausführen kann. SIBs werden nicht über Package- und Klassennamen angesprochen, sondern über

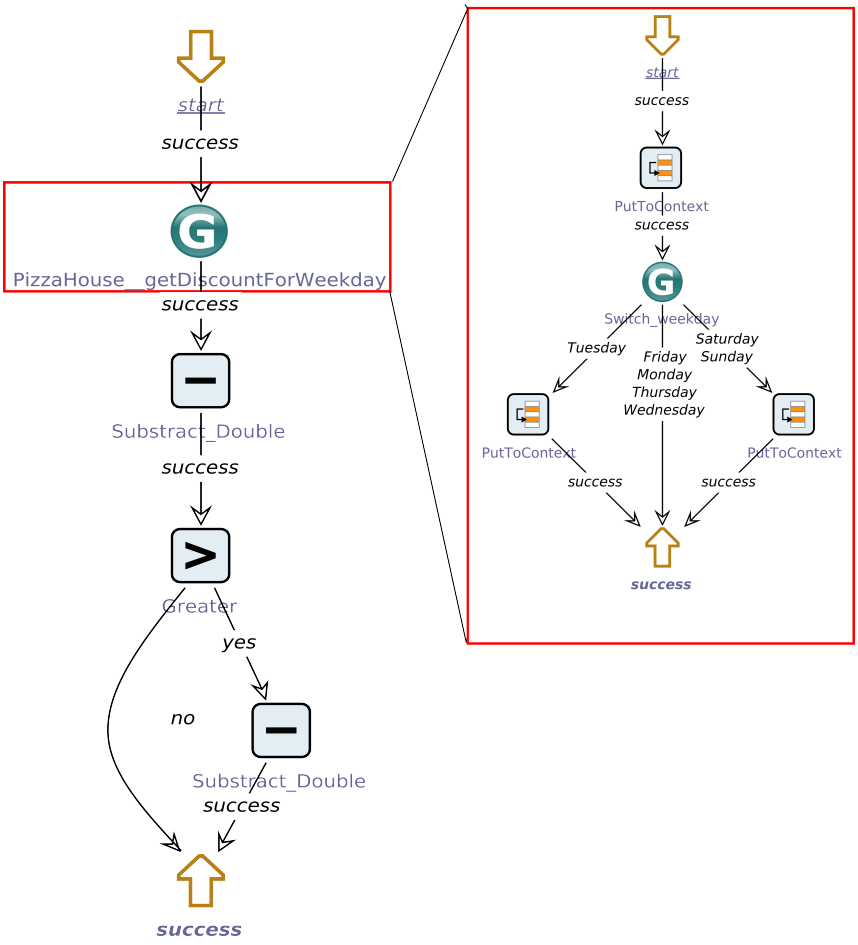


Abbildung 2.4: Graph-Hierarchie in jABC

eine eindeutige UID. Ändert sich die Klassenname oder die Projektstruktur, bleibt die Referenz auf das SIB erhalten [39]. Ein SIB besteht aus:

- Eingabe-Parametern, mit deren Hilfe das SIB konfiguriert werden kann,
- ausgehenden Kanten (*Branches*), die die möglichen Ergebnisse einer Ausführung zeigen,
- einem Symbol bzw. Icon und einem Namen, um das SIB im Editor darzustellen, sowie
- einer Dokumentation des Knotens selbst, seiner Parameter und Kanten [61].

SIBs können über einen gemeinsamen Speicher, auch *Kontext* genannt, untereinander Informationen austauschen. Jedes SIB ist in der Lage Daten aus dem Kontext zu lesen, zu verändern oder neu abzulegen. Dieser ist nach dem *Blackboard*-Architekturmuster realisiert und besteht aus einer Hash-Tabelle, die sich aus Name-Wert-Paaren aufbaut. Jedes Datenelement, welches in Kontext abgelegt wird, besitzt damit einen eindeutigen Namen (auch *Kontextvariable* genannt) und einen Wert. Letzterer kann durch einen einfachen Datentyp oder ein komplexes Objekt repräsentiert sein [39]. Die Daten werden auf dem Blackboard von einzelnen SIBs in einer hierarchisch organisierten Form abgelegt. Das Blackboard ist in der Lage, andere SIBs von der Ablage oder Änderung dieser Daten zu benachrichtigen.

Die eigentliche Implementierung eines SIBs wird durch den SIB-Experten entwickelt und hängt von der Version des jABC ab. Bis zur Version 3.x muss die Funktionalität eines SIBs manuell implementiert werden. Dazu muss die Methoden `execute` im SIB-Interface umgesetzt werden. Um eine Entkopplung zwischen dem Modell und der Implementierung des SIBs zu ermöglichen, kann der *ServiceAdapter* verwendet werden. Bei diesem Muster wird in der `execute`-Methode der Adapter aufgerufen, welcher die eigene Funktionalität enthält. Als Adapter können beliebige Klassen in beliebigen Programmiersprachen dienen. Ab der Version 4 können SIBs öffentliche Methoden der Klassen im Classpath direkt aufrufen, diese werden als *Service-Calls* bezeichnet [40]. SLGs werden mithilfe der Service-Calls und anderer SLGs zusammengesetzt.

In allen Versionen ist die Implementierung der Funktionalität eines SIBs ist im Framework verborgen und für Anwendungsexperten nicht erkennbar. Das sollte den Experten nicht stören, denn für die Modellierung ist lediglich das intendierte Verhalten des SIBs wichtig. Der Fachexperte ist für das richtige Verwenden der SIBs und Anpassung der Ein- und Ausgabeparametern zuständig [34].

2.5.2 Editor

jABC stellt eine grafische Oberfläche zur Modellierung von Prozessgraphen zur Verfügung. Dazu gehören die Zeichenfläche zur Erstellung und Anpassung der Modelle, eine Projektverwaltung, eine Bibliothek mit Prozessbausteinen sowie verschiedene Erweiterungen, die die Semantik der Modelle definieren. Abbildung 2.5 zeigen die wichtigsten Komponenten des Editors. Die Projektverwaltung (Bereich A) enthält eine Liste aller verfügbaren Projekte. Jedes Projekt setzt sich aus Prozessgraphen und Prozessbausteinen zusammen. Die Eigenschaften von SLGs oder SIBs können über die Inspektoren (Bereich B) unten links eingesehen und verändert werden. Dazu gehören beispielsweise der Name und der Typ des

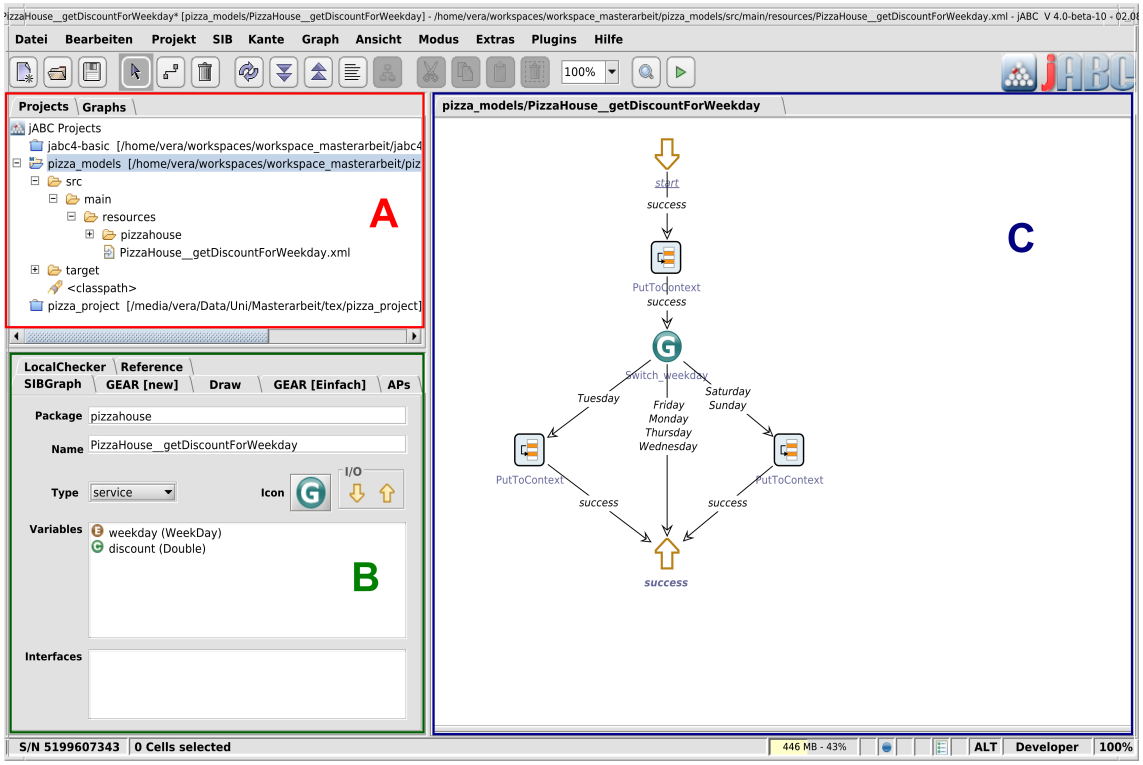


Abbildung 2.5: jABC-Editor

Prozessgraphen, die Namen und Typen von Kontextvariablen.

Im rechten Teil Mitte befindet sich das Hauptfenster der Anwendung. Auf der Zeichenfläche (Bereich C auf Abbildung 2.5) kann der Nutzer einen Prozessgraphen erstellen. SIB-Knoten können direkt aus dem SIB-Baum (Abbildung 2.6) ausgewählt und per Drag-And-Drop dem Modellgraphen hinzugefügt werden. Jeder Prozessgraph wird in einem eigenen Fenster dargestellt, im Editor sind diese durch Karteireiter getrennt.

2.5.3 Plugin-Architektur

Durch seine Plugin-Architektur kann das jABC einfach um neue Funktionen erweitert werden. Plugins im jABC sind Java-Klassen, die ein vorgegebenes Interface implementieren müssen. Durch Plugins können neue Funktionen hinzugefügt oder zusätzliche Semantiken für Modelle definiert werden. Des Weiteren können Plugins für ihre Konfiguration eigene Menüs und Inspektoren erstellen und diese in die grafische Oberfläche des jABCs einfügen [39]. Plugins haben Zugriff auf die Elemente der graphischen Oberfläche, die Projektverwaltung sowie alle Prozessgraphen. Folgende Plugins werden zusammen mit dem Framework mitgeliefert und sind bei der Modellierung sehr hilfreich [40].

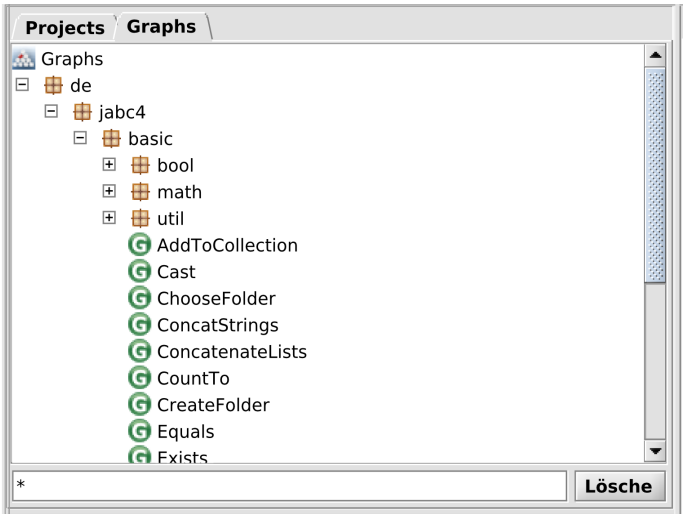


Abbildung 2.6: SIB-Baum Inspektor

- Das **Tracer**-Plugin kann Modelle interpretieren und ausführen. Ein Anwender kann die Ausführung manuell oder durch Haltepunkte unterbrechen.
- Das Prüfen von lokalen Eigenschaften eines SIBs wird durch das **LocalChecker**-Plugin vorgenommen. Geprüft werden zum Beispiel Ein- und Ausgabeparameter eines SIBs sowie die Namen seines ausgehenden Branches. Falls Fehler oder Warnungen auftreten, erfolgt eine visuelle Rückmeldung im Editor.
- Die Aufgabe des **ModelChecker**-Plugins ist die Verifikation von globalen Eigenschaften eines jABC-Modells. Dazu kann der Anwender verschiedene temporallogische Formel hinterlegen, die der ModelChecker dann anhand des vorliegenden Modells auswertet. Wie beim LocalChecker erfolgt eine visuelle Fehlermeldung, falls das Modell die Formel nicht erfüllt.
- Das **Layouter**-Plugin besitzt mehrere Algorithmen zur automatischen Auslegung der SIBs im Graphen.

2.5.4 jABC v.4

In der Version 4 des jABC wird die manuelle Implementierung der einzelnen SIBs durch hierarchische SLGs ersetzt. Ein Modellelement in einem SLG kann entweder auf eine öffentliche Methode einer bekannten Java-Klasse oder auf einen anderen SLG verweisen. Die direkten Aufrufe von Methoden werden als *Service-Calls* bezeichnet und besitzen die Ein- und Ausgabeparameter der aufrufenden Methode. Ein Modell beginnt immer mit einem Start-SIB, welcher auch *Input-SIB* genannt wird. Modelle können ein oder mehrere Ausgabe-Branches haben, abhängig von der Logik, die im Modell implementiert ist. Die Ausgabe-Branches werden auch Modell-Branches genannt und in *Output-SIBs* umgesetzt. Wenn die Ausführung eines Modells eine Ausnahme werfen kann, können die Ausgabe-Branches *success* und *exception* heißen. Je nach dem, ob die Ausführung erfolgreich was oder eine Ausnahmesituation aufgetreten ist, wird der jeweilige Branch zurückgegeben. Ein Modellgraph, welcher auf eine Frage mit „Ja“ oder „Nein“ antwortet (wie z.B. „Ist eine Kollektion leer?“), erhält die Branches *yes* und *no*. Somit ergeben sich folgende vier Arten der SIBs:

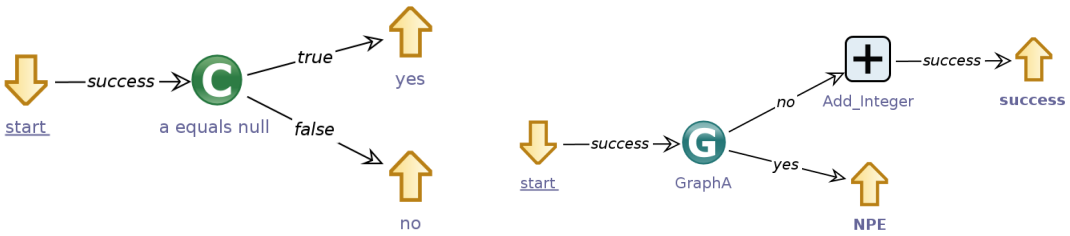
Input-SIBs definieren die Eingabe-Parameter eines Modellgraphens.

Output-SIBs repräsentieren die möglichen Ergenisse der Ausführung eines Graphens.

Service-Calls rufen direkt eine in diesem Projekt verfügbare Java-Methode auf.

Service-Graphs sind SLGs, besitzen Eingabe-Parameter und Modell-branches und können in anderen SLGs aufgerufen werden.

Die verschiedenen SIBs werden in Abbildung 2.7 veranschaulicht. Mithilfe des Graphen A kann entschieden werden, ob ein Objekt gleich `Null` ist oder nicht. Der Graph B führt eine Nullpointer-sichere Berechnung durch und benutzt dabei den Graphen A. Der Graph A besitzt zwei Ausgabe-Branches *yes* und *no*. Diese werden im Graphen B als Kanten-Markierungen verwendet.



(a) Graph A **(b)** Graph B

Abbildung 2.7: Beispiele für jABC4-Graphen

Um Routineaufgaben zu lösen, bringt das jABC bereits eine große Menge an fertigen Prozessknoten mit, die elementare Operationen implementieren. Abbildung 2.8 zeigt einige SIBs aus der Kollektion.

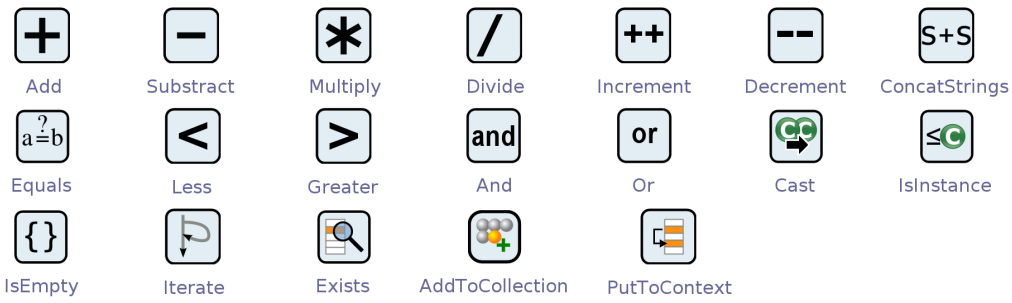


Abbildung 2.8: Einige SIBs aus der Basic-Kollektion

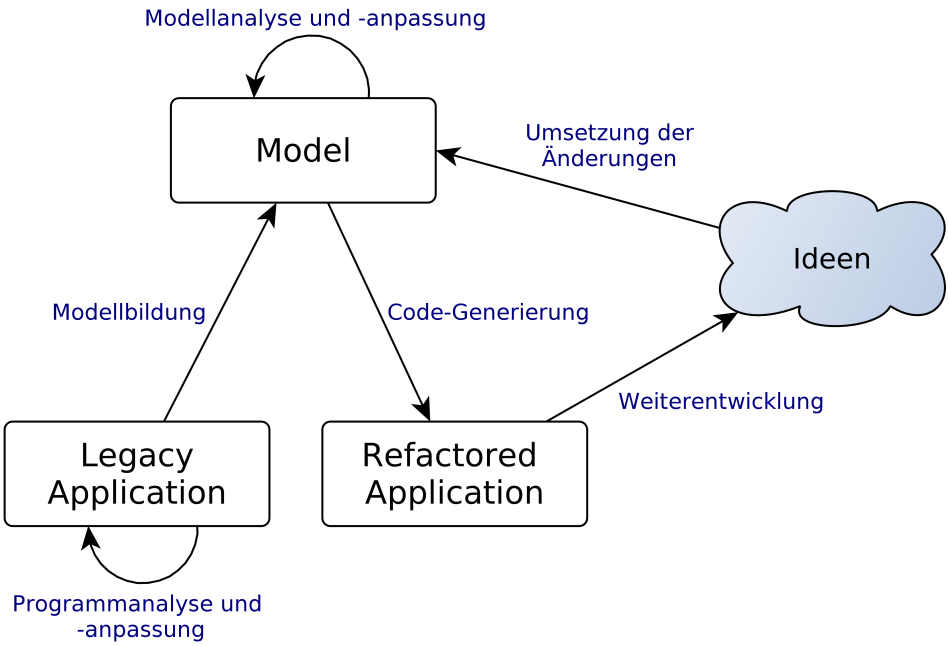


Abbildung 2.9: Modellgetriebenes Reengineering (vgl. [52])

2.6 Modellgetriebenes Reengineering

Während im klassischen Reengineering Modelle dem Anwendungsverstehen dienen und bei der Planung der Neuimplementierung unterstützen, werden diese beim modellgetriebenen Reengineering als Ausgangsbasis für das neue System erstellt. Änderungen oder Erweiterungen an einer Software finden jetzt ausschließlich auf der Modellebene statt. Modelle werden damit zu zentralen Artefakten des neuen Systems. Modellgetriebenes Reengineering kann in mehrere Phasen unterteilt werden (vgl. [61]). Diese werden in Abbildung 2.9 visualisiert.

2.6.1 Programmanalyse und -anpassung

Im ersten Schritt wird der vorhandene Programmcode analysiert und in ein oder mehrere Modelle übersetzt. Bei der Analyse muss zum Beispiel entschieden werden, welche Teile des Systems durch Reengineering verändert werden. Wenn die Modelle automatisch erstellt werden, müssen die ausgewählten Teile des Programmcodes unter Umständen durch Refactoring angepasst werden. Dies setzt natürlich voraus, dass die bestehende Software vollständig als Quellcode vorliegt.

2.6.2 Modellbildung

Nach der Analyse wird der Quellcode in eine sprach- und plattformunabhängige Beschreibung transformiert. Diese Beschreibung enthält Informationen, die sowohl implizit als auch explizit im Quellcode vorhanden sind. Beispielsweise können aus dem Quellcode abstrakte syntaktische Bäume aufgebaut werden, welche Programmiersprachenkonstrukte in vordefinierte Kategorien klassifizieren. Im Anschluss daran wird die abstrakte Beschreibung von einem Modellierungstool eingelesen und grafisch dargestellt. Dazu werden alle vorhandenen Informationen zu jedem Programmiersprachenkonstrukt und dessen Beziehungen

in die interne Repräsentation des Modellierungswerkzeugs überführt. Die daraus resultierenden Modelle werden aufgrund ihrer Nähe zum Programmcode auch *Code-Modelle* genannt [61]. Die Art der Modelle hängt vom ausgewählten Werkzeug ab. Zum Beispiel können Modelle den Kontrollfluss des Programmcodes darstellen. Andere Art der Modelle können Beziehungen zwischen Programmelementen repräsentieren, wie beispielsweise der Aufruf-Graph. Für UML-ähnliche Modelle können die Beziehungen zwischen Klassen und den Klassenvariablen in Modelle miteinfließen.

2.6.3 Modellanalyse und -anpassung

Nach der Erstellung der Modelle können diese mit Hilfe verschiedener statischer Analysen untersucht werden. Die Modellanalyse beinhaltet unter anderem auch eine Validierung der erstellten Modelle und eine Fehlerkorrektur. Während MDSD-Frameworks üblicherweise syntaktische Validierung mit sich bringen, können fachliche Fehler nicht automatisch erkannt werden [34]. Um die abgebildeten Fachlichkeiten auf Korrektheit zu überprüfen, wird Wissen der Fachexperten benötigt. Zur Bewertung der Qualität von erstellten Modellen können bestimmte Metriken gemessen werden, wie z.B. zyklomatische Komplexität, Anteil von ungenutztem Programmcode, Abhängigkeiten zwischen Komponenten und andere. Die Messungen können entweder manuell stattfinden, oder mit Hilfe neuer Modelle. Vorteilhaft ist die modellgetriebene Messung, wenn die Anzahl der Modelle groß ist oder wenn die Ergebnisse der Auswertung visualisiert werden müssen. Die gemessenen Kennzahlen werden zur späteren Optimierung bzw. Anpassung der Modelle benutzt.

2.6.4 Code-Generierung und Verifikation

Nach der Modellanalyse wird mit dem Code-Generator aus den erzeugten Modellen Programmcode generiert. Dadurch entsteht eine neue Version des Originalsystems. Das Verhalten des neuen Systems muss dem Verhalten des alten Systems entsprechen. Es muss also überprüft werden, ob das alte und das neue System die selbe Funktionalität besitzen. Um festzustellen, dass das Verhalten zweier Systeme gleich ist, können verschiedene Techniken angewandt werden. Man kann den Code-Generator so entwerfen, dass dieser den ursprünglichen Code Zeile für Zeile reproduziert. Nach der Code-Generierung können der Programmcode beider Systeme einfach verglichen werden. Eine Alternative dazu bildet der Einsatz von einem Lernverfahren, das das Verhalten des alten Systems lernt. Beim Lernen stellt ein Automat Anfragen an das gewählte System, bekommt die Antworten auf diese Anfragen und versucht das Verhältnis zwischen Ein- und Ausgaben des Systems in einer Funktion abzubilden. Danach stellt der Lernautomat die gleichen Anfragen an das neue System und überprüft, ob die Antworten die gleichen sind. In der Praxis wird häufig die Gleichheit der Systeme durch Tests überprüft. Die Tests für das alte System müssen auf dem neuen System die gleichen Ausgaben produzieren können. Dann wird davon ausgegangen, dass durch Reengineering das Verhalten erhalten blieb.

2.6.5 Weiterentwicklung des neuen Systems

Für die Teile des Systems, die durch Reengineering in Modelle überführt wurden, findet die Weiterentwicklung auf der Modellebene statt. Sobald eine Änderung bzw. neue Komponente als Idee vorliegt, können diese in Modellen umgesetzt werden. Die neuen Modelle werden wie gewohnt mit Hilfe des Code-Generators in Quellcode übersetzt.

3 Ansatz und Realisierung

Modellgetriebenes Reengineering ist ein Prozess, der in mehrere Phasen unterteilt werden kann. Der Fokus dieser Masterarbeit liegt auf der Code-zu-Modell-Transformation, auch *Modellbildung* genannt. Die restlichen Phasen werden entweder durch manuelle Anpassungen oder mit Hilfe existierenden Werkzeuge durchgeführt. In diesem Kapitel wird der im Rahmen der Masterarbeit entwickelte Ansatz zur semiautomatischen Modellbildung vorgestellt. Obwohl die prototypische Implementierung nur Java-Code in Modelle übersetzen kann, ist der Ansatz auf andere Programmiersprachen übertragbar.

Bei der Modellbildung wird der existierende Quellcode in eine plattformunabhängige Repräsentation transformiert. Im Anschluss daran wird diese Repräsentation von einem Modellierungstool eingelesen und grafisch dargestellt. Anhand vieler Bespiele wird erklärt, wie einzelne Code-Elemente in Modell-Elemente transformiert werden. Da eine Betrachtung aller Java-Konstrukte den Zeitrahmen der Arbeit sprengen würde, wird das Verfahren auf eine bestimmte Untermenge der Code-Elemente eingeschränkt. Es wird außerdem diskutiert, wie sich die Korrektheit der Transformation überprüfen lässt.

3.1 Ansatz

Es existieren zwei Ansätze zur Entwicklung der Modelle aus Quellcode: Top-Down und Bottom-Up [31]. Eine Kombination von beiden Vorgehensweisen ist ebenfalls möglich (vgl. [38]). Der Top-Down-Ansatz setzt gutes Verständnis der zugrundeliegenden Geschäftsprozesse voraus. Dieser Ansatz beginnt mit Erstellung eines Modells des Gesamtsystems, welches das Verhalten des Systems darstellt. Danach wird die zunächst sehr abstrakt modellierte Funktionalität des Systems möglichst sinnvoll gegliedert. In den nachfolgenden Schritten werden die definierten Subsysteme auf der jeweils nächsten untergeordneten Hierarchieebene in Modellen abgebildet, bis der angestrebte Detailgrad erreicht wird [31]. Anschließend werden die Modelle mit Komponenten des Software-Systems verknüpft. Dabei kann für jedes Modellelement eine existierende Komponente als Implementierung hinzugefügt oder neu realisiert werden. Von Vorteil ist das Top-Down-Vorgehen bei übersichtlichen Systemen bzw. bei Reengineering von handhabbaren Teilsystemen. Problematisch ist jedoch, dass auf einer höheren Abstraktionsstufe noch nicht feststellbar ist, ob eine geplante Komponente sich tatsächlich in der Realisierung finden oder implementieren lässt.

Um diese Problematik zu vermeiden, kann der Bottom-Up-Ansatz verfolgt werden. Bei dieser Vorgehensweise wird die Modellierung auf der untersten Detailebene, also mit dem Quellcode, begonnen. Die Basiskomponenten (wie z.B. Anweisungen im Code, Methoden oder ganze Klassen) werden identifiziert und in feingranulare Modellelemente transformiert. In weiteren Schritten werden die Basiselemente zu immer komplexeren Funktions-

einheiten zusammengesetzt, bis eine hohe Abstraktionsebene erreicht wird. Nachteilig bei dieser Herangehensweise ist, dass nicht a priori festgestellt werden kann, ob eine Integration der einzelnen Komponenten zu einem angestrebten Gesamtsystem möglich ist [31]. Außerdem besteht die Gefahr der Duplikationen, wenn gleiche Teilkomponenten in verschiedenen Untersystemen nicht erkannt und zusammengefasst wurden. Der Bottom-Up-Entwurf dient deshalb meist dazu, Funktionseinheiten als Services in einer Bibliothek bereitzustellen. Die Prozesse, welche diese Services verwenden, müssen oft manuell nachgearbeitet werden.

In dieser Arbeit wird ein semiautomatischer Ansatz zur Modellbildung vorgestellt. Dieser wird im Folgenden als *Code-Importer* bezeichnet. Mithilfe des Code-Importers kann entweder das Gesamtsystem oder nur einzelne Teile in jABC-Modelle übersetzt werden, z.B. alle Prozesse in einem bestimmten Modul. In einer Enterprise-Applikation mit ausgereifter Schicht-Architektur scheint das Reengineering eines geschäftslogischen Moduls natürlicher als Reengineering des Gesamtsystems. Dabei können die technischen Schichten wie Datenpersistenz- oder Oberflächenprogrammierung unverändert bleiben.

Der Code-Importer verwendet eine Kombination aus zwei Ansätzen zur Modellbildung. Ähnlich wie der Bottom-Up-Ansatz setzt der Code-Importer auf der Quellcode-Ebene an und erstellt die sogenannten Code-Modelle. Allerdings wird bei der Erstellung der Modelle in Top-Down-Richtung vorgegangen. In einer existierenden Applikation werden Geschäftsprozesse oder Teile davon typischerweise in Methoden abgebildet. Die einzelnen Schritte eines Prozesses können wiederum als Methoden definiert und beliebig komplex sein.

Der Modellierungsvorgang beginnt mit einer vom Benutzer ausgewählten Methode, die den gewünschten Geschäftsprozess vollständig abbildet. Diese wird im Folgenden als *Startmethode* genannt. Die Startmethode wird in Top-Down Richtung zu einem Modell transformiert: Die Anweisungen werden zu Modellelementen und entsprechend dem Kontrollfluss durch Kanten verbunden. Es werden zwei Arten der Modellelemente in jABC4 unterschieden. Elementare Einheiten implementieren eine eingegrenzte Funktionalität, die nicht weiter zerlegt werden kann. Die zweite Art der Modellelemente sind Untermodelle. Diese können selbst aus elementaren Einheiten oder weiteren Untermodellen bestehen. Bei Methoden, die in der Startmethode aufgerufen werden, muss definiert werden, ob diese als elementare Funktionalitäten oder als Untermodelle betrachtet werden. Denn die Übersetzung dieser Methoden hängt von deren Semantik ab. Typischerweise werden in der Applikation komplexe Vorgänge in Methoden implementiert und in einer übergeordneten Methode referenziert. Da Startmethode einen Prozess abbildet, werden einige Unterprozesse in weiteren Methoden implementiert. Diese Methoden sollten bei der Modellbildung als Untermodelle repräsentiert werden. Aufrufe einer externen Bibliothek (z.B. Logging, Datenbankverwaltung usw.) sind wahrscheinlich nicht Teil des Geschäftsprozesses sondern stellen andere Aspekte des Systems dar. Diese sollten als elementare Modelleinheiten (wie beispielsweise Service-Calls) abgebildet werden. Zu den elementaren Modelleinheiten zählen außerdem Operatoren wie + oder −, für die existierende SIBs aus der *Basic*-Bibliothek (s. Abschnitt 2.5.4) verwendet werden.

Der Code-Importer muss für jeden Methodenaufruf entscheiden, ob für diesen ein Service-Call oder ein Untermodell erzeugt wird. Dafür steht ihm die sogenannte *Top-Down-*

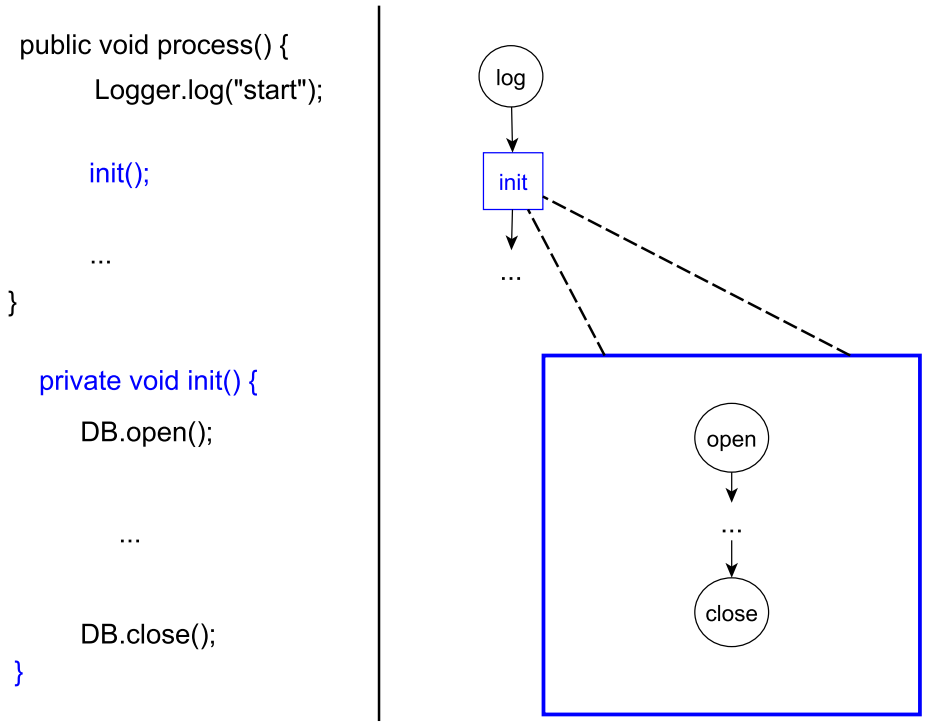


Abbildung 3.1: Eine Top-Down-Strategie

Strategie zur Verfügung. Diese umfasst eine Menge von Regeln. Die Regeln beschreiben, wann ein Methodenaufruf in eine elementare Einheit transformiert und wann ein Untermodell erstellt wird. Abbildung 3.1 veranschaulicht eine Top-Down-Strategie, die nur die Methoden der gleichen Klasse als Untermodelle definiert.

Weitere Bespiele für mögliche Top-Down-Strategien werden im Folgenden aufgeführt:

- Alle Methoden der gleichen Klasse oder der Klasse aus der gleichen Vererbungshierarchie werden als Untermodellen dargestellt. Sonst werden Service-Calls erstellt.
- Methoden aus Klassen im gleichen Modul, wie die aktuelle Klasse, werden zu Untermodellen. Sonst werden Service-Calls erstellt.
- Alle Methoden aus der aktuellen Applikation werden zu Untermodellen. Für externe Methoden werden Service-Calls erstellt.

Der Code-Importer kann um benutzerspezifische Top-Down-Strategien erweitert werden. Dazu muss lediglich das Interface `TopDownStrategie` implementiert und der Konfigurations-GUI hinzugefügt werden. Außerdem können die existierenden Strategien angepasst bzw. ausgetauscht werden.

3.2 Realisierung

Der Code-Importer wird als jABC-Plugin realisiert, um aus dem gegebenen Java-Quellcode ausführbare Service-Logic-Graphen zu erstellen. Mit Hilfe des Plugins können mehrere

Methoden aus mehreren Klassen in Modelle übersetzt werden. Der Übersetzungsvorgang besteht aus mehreren Schritten. Zunächst wird die Konfiguration eingelesen, welche unter anderem die Liste der zu übersetzenden Methoden und die zu benutzende Top-Down-Strategie enthält. Danach wird mittels eines Java-Parsers der Quellcode eingelesen und in abstrakte Syntaxbäume übersetzt. Jeder AST wird in einen plattformabhängigen Kontrollflussgraphen transformiert. Aus jedem Kontrollflussgraphen wird je ein SLG aufgebaut. Falls einer der Schritte mit einem Fehler endet, wird der Gesamtvorgang abgebrochen und eine Fehlermeldung angezeigt. Eine mögliche Ursache für einen Fehlerzustand ist z.B. der Versuch eine Quelltextdatei zu parsen, die nicht den Anforderungen des Code-Importers entspricht (s. Abschnitt 3.4). Die einzelnen Schritte werden in Abbildung 3.2 veranschaulicht und im Folgenden ausführlich erklärt.

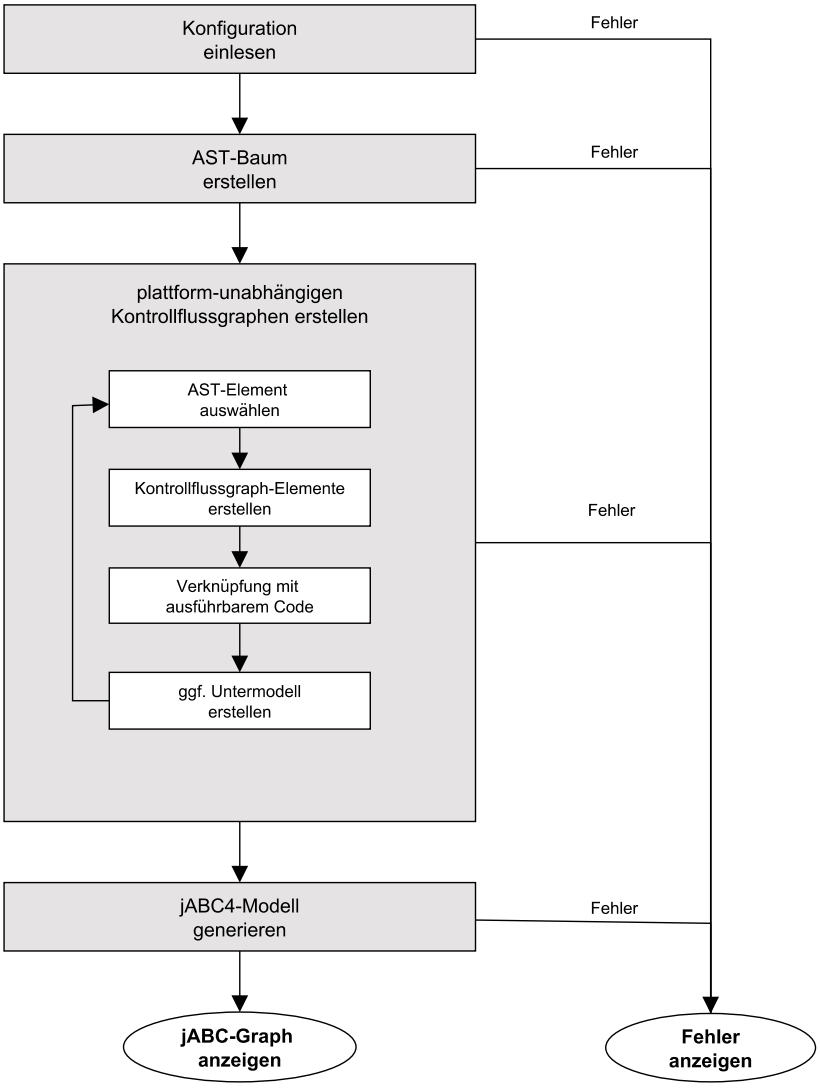


Abbildung 3.2: Aufbau eines Modells

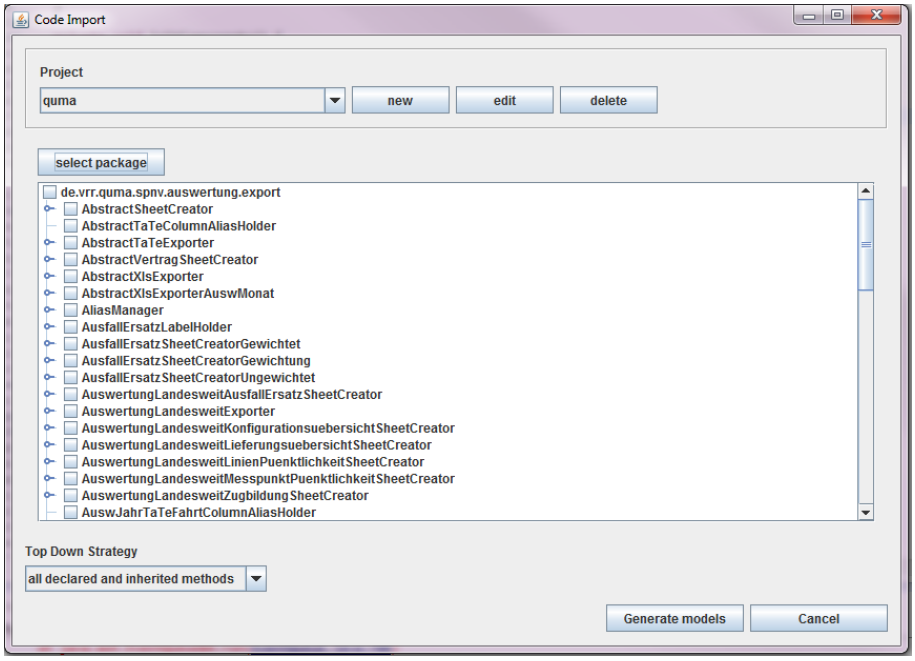


Abbildung 3.3: Grafische Oberfläche des Plugins Code-Importer

Einlesen der Konfiguration

Zu dem jABC-Plugin für den Code-Importer gehört die Konfigurations-GUI, die in Abbildung 3.3 dargestellt ist. Diese ermöglicht die Auswahl des Projektes, das durch Reengineering verändert wird. Nachdem der Benutzer ein Paket aus dem Projekt ausgewählt hat, kann er eine oder mehrere Methoden aus verschiedenen Java-Klassen ankreuzen. Die markierten Methoden werden in Modelle transformiert und im Editor angezeigt. Die GUI bietet die Auswahl der Top-Down-Strategie an, welche die Tiefe der Graphenhierarchie bestimmt.

Im ersten Schritt der Transformation wird die Konfiguration eingelesen. Diese umfasst folgende Informationen:

- Pfad zum Quellcode und zu den kompilierten Dateien
- Pfad zum jABC-Projekt, in dem die Modelle gespeichert werden
- Package- und Klassenname der zu übersetzenden Java-Klasse
- eine Liste der Methoden, die in Modelle umgewandelt werden
- eine Top-Down-Strategie

Konstruktion des AST

Der eigentliche Übersetzungsvorgang beginnt mit dem Einlesen des Quellcodes und der Konstruktion eines abstrakten Syntaxbaumes (kurz: AST). Für die Erzeugung des AST wird das Tool *JavaParser* [26] in der Version 1.0.8 verwendet. Der Parser basiert auf dem *Visitor*-Muster [17] und generiert den AST für jede Methode in der gegebenen Klasse. Außerdem enthält das Ergebnis des Parsers die Klassenattribute, Import-Anweisungen

und die Namen der Oberklassen. Der generierte AST besteht aus folgenden Knotentypen: `TypeDeclaration`, `MethodDeclaration`, `Statement` und `Expression`. Der Syntaxbaum einer Methodendeklaration wird in einer Instanz der Klasse `MethodDeclaration` festgehalten. Dabei wird der Rumpf einer Methode als Blockanweisung, bestehend aus mehreren Anweisungen (`Statement`), dargestellt. Die einfachste Art einer Anweisung ist ein Ausdruck (`Expression`). Ausdrücke lassen sich in Operatoren, Variablennamen bzw. Literale zerlegen.

Im Folgenden wird am Beispiel einer `If`-Anweisung der Aufbau des AST erklärt. In Listing 3.1 wird eine `If`-Anweisung dargestellt, die mit Hilfe des Parsers verarbeitet wird.

Listing 3.1: If-Anweisung

```
if (size > 0){
    doSomething(collection);
}
else{
    doSomethingElse();
}
```

Die Bedingungsanweisung aus Listing 3.1 wird als `IfStatement` repräsentiert. Ein If-Statement besteht aus einem binären Ausdruck (die Bedingung), einer `Then`-Blockanweisung und einer `Else`-Blockanweisung. Der binäre Ausdruck `size > 0` lässt sich in den Operator `>`, den linken Ausdruck und den rechten Ausdruck zerlegen. Der linke Ausdruck stellt einen Variablennamen dar, der rechte Ausdruck ist ein Integer-Literal. Die `Then`- und die `Else`-Blockanweisungen sind Ausdrucksanweisungen vom Typ `MethodCallExpression`. Jeder Methodenaufruf besitzt einen Methodennamen, eine Liste von Argumenten und gegebenfalls das Objekt, auf dem die Methode ausgeführt wird (*Scope*). Abbildung 3.4 veranschaulicht den AST für das Beispiel aus Listing 3.1. Dabei werden alle AST-Elemente vom Typ `Statement` grün und die Ausdrucksanweisungen vom Typ `Expression` rosa markiert.

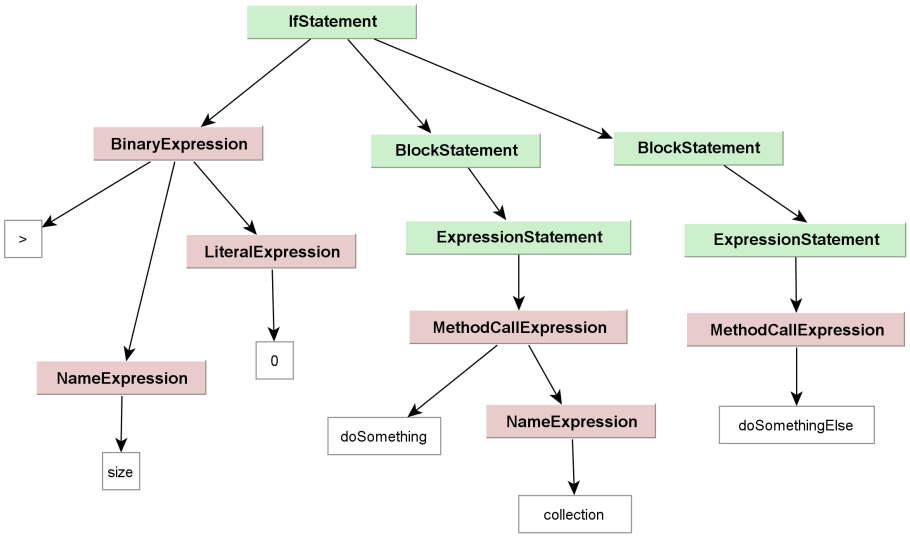


Abbildung 3.4: Abstrakter Syntaxbaum zur If-Anweisung

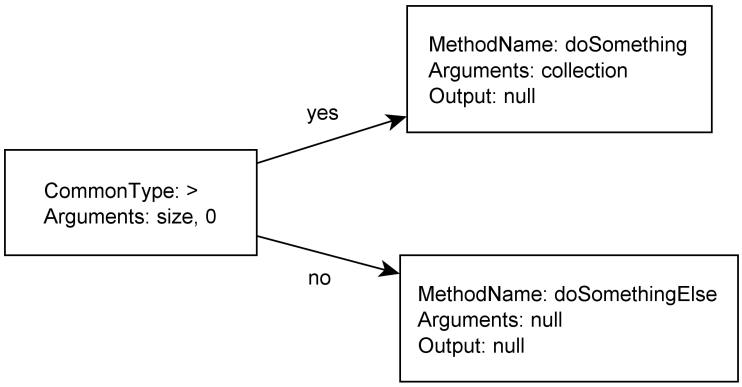


Abbildung 3.5: Kontrollflussgraph-Elemente zur If-Anweisung

Plattformunabhängiger Kontrollflussgraph

Im nächsten Schritt wird aus dem abstrakten Syntaxbaum der plattformunabhängige Kontrollflussgraph erstellt. Dieser Graph ähnelt dem Service Logic Graph, welcher in jABC verwendet wird. Der Graph besteht aus einem Startknoten, besitzt beschriftete Kanten und einen oder mehrere Ausgabe-Knoten. Jeder Knoten erhält den Namen der Methode bzw. den Typ der Operation, eine Liste der Argumente sowie einen Rückgabewert.

Die Übersetzung eines AST in einen Kontrollflussgraphen basiert auf dem *Interpreter*-Muster [17]. Für jeden AST-Knotentyp wird ein eigener Interpreter entwickelt, der den zugehörigen Teilbaum in Teile des Kontrollflussgraphen transformiert. Ein AST-Knoten kann in einen Kontrollflussknoten, einen Teilgraphen (mehrere Knoten mit Kanten), eine Kante oder eine Eigenschaft eines Knotens übersetzt werden. Darauf, wie die einzelnen Java-Konstrukte bzw. AST-Elemente behandelt werden, wird im Abschnitt 3.3 eingegangen. Die für die `If`-Anweisung aus Listing 3.1 generierten Kontrollflussgraph-Elemente werden in Abbildung 3.5 dargestellt.
Die Elemente des Kontrollflussgraphen werden anschließende mit dem ausführbaren Code verknüpft. Dazu werden zu den Knotenbeschreibungen Referenzen auf die existierenden plattformspezifische Implementierungen hinzugefügt. Ein Service-Call in jABC benötigt die Referenz auf die Instanz von `java.lang.reflect.Method` sowie die Referenz auf die deklarierende Klasse vom Typ `java.lang.Class`. Anhand des Methodenaufrufs wird nach den zugehörigen `Method` und `Class` wie folgt gesucht.

1. Identifiziere die Klasse, in der die gesuchte Methode enthalten ist.

2. Suche in der Klasse nach einer Methode mit dem gleichen Namen und gleicher Argumentenanzahl. Wenn sich alle deklarierten Argumententypen von den gegebenen Typen ableiten lassen, gib die Methode zurück.

3. Sonst suche die Oberklasse der aktuellen Klasse. Wenn es eine Oberklasse existiert, gehe zu Schritt 2. Ansonsten gib eine Fehlermeldung aus.

Wenn die passende Methode gefunden wird, wird der Kontrollflussgraph um die Informationen *Class* und *Method* ergänzt. Der aktualisierte Kontrollflussgraph zur `If`-Anweisung wird in Abbildung 3.6 dargestellt.

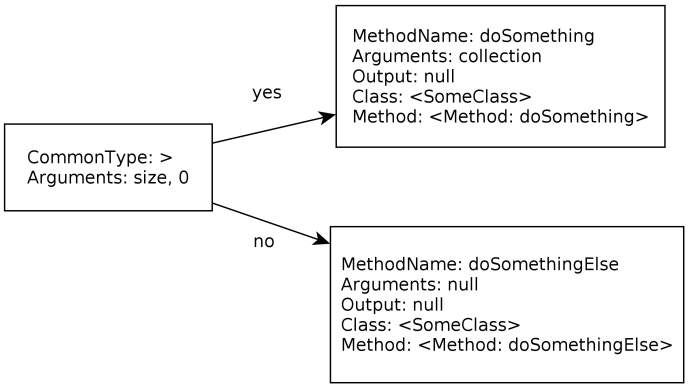


Abbildung 3.6: Kontrollflussgraph mit Class und Method

Ausgehend von der Top-Down-Strategie werden ggf. Untermodelle erzeugt. Eine Top-Down-Strategie implementiert die Funktion, die einen beliebigen Methodenaufruf auf $\{true, false\}$ abbildet. Wenn die Strategie *true* ausgibt, wird die Methode in einen Untergraph umgewandelt. Top-Down-Strategien können bestimmte Informationen, wie Methodenname, die deklarierende Instanz (Scope) und die Methodenargumente, verwenden. Folgendes Beispiel zeigt eine mögliche Strategie und deren Umsetzung:

Strategie	Methoden der gleichen Klasse werden als Untermodelle repräsentiert.
Implementierung	Wenn der Scope leer ist oder `this` heißt, wird *true* zurückgegeben. Sonst wird $false$ ausgegeben.

Wenn die Strategie *true* zurückgibt, wird die Methodendeklaration in einen Untergraphen überführt. Dazu werden Schritte 2 bis 4 für diese Methode wiederholt. Das bedeutet, der neue AST-Baum und der neue Kontrollflussgraph mit weiteren Untergraphen muss erstellt werden. Der rekursive Prozess bricht ab, wenn in einem Untergraphen lediglich Service-Calls zu finden sind. Für die `If`-Anweisung aus Listing 3.1 wird eine Top-Down-Strategie verwendet, die beide Methoden `doSomething` und `doSomethingElse` als Untergraphen übersetzen lässt. Der komplette Kontrollflussgraph für die Anweisung wird in Abbildung 3.7 gezeigt.

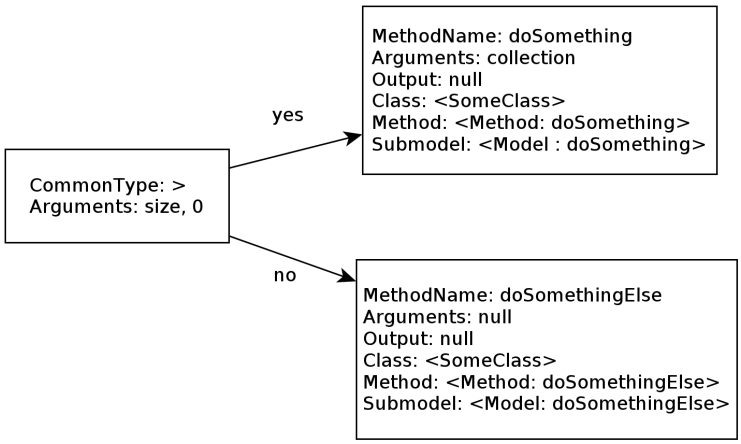


Abbildung 3.7: Plattformunabhängiger Kontrollflussgraph für die If-Anweisung

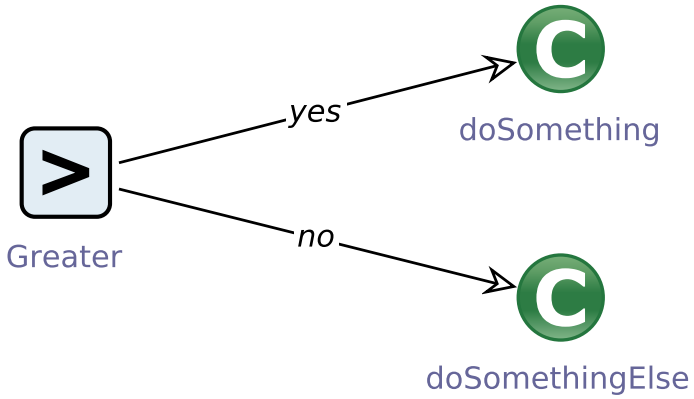


Abbildung 3.8: Das jABC-Modell zur If-Anweisung

Erstellen des jABC-Modells

Die letzte Phase der Modellbildung umfasst die Erstellung eines jABC-Modells. In diesem Schritt werden aus den Knoten des Kontrollflussgraphen Instanzen folgender SIBs. Der Startknoten wird zu einem `InputSIB`, die Ausgabe-Knoten werden in `OutputSIBs` überführt. Für alle anderen Knoten wird entweder ein `ServiceCall` oder ein `ServiceGraph` erstellt. Service-Graphen werden erzeugt, wenn der Knoten einen Untermodell besitzt oder wenn der Graph einen Graphen aus der *Basic*-Bibliothek (s. Abschnitt 2.5.4) referenziert. Die Argumente aller Knoten werden als Kontextvariablen dargestellt und dem Kontext hinzugefügt. Für jede Kante wird eine Instanz von `SibGraphEdge` erzeugt. Abbildung 3.8 zeigt das finale jABC-Modell für das Beispiel aus Listing 3.1.
Damit die Modelle ausführbar sind, wird sichergestellt, dass unter anderem folgende Bedingungen erfüllt werden:

- alle Modelle besitzen genau ein Start-SIB
- alle Kanten sind korrekt beschriftet
- alle SIBs sind vom Start-SIB erreichbar
- alle Kontextvariablen, die gelesen werden, sind entweder Parameter des Input-SIBs oder werden als Ausgabe eines anderen SIBs im vorgelagerten Teil des Graphen erzeugt
- bei allen Service-Calls mit nicht-statischen Methoden muss die Referenz auf die Instanz korrekt gesetzt sein
- alle in Service-Calls referenzierten Methoden müssen im Classpath sichtbar sein
- alle Untergraphen müssen korrekt referenziert sein
- alle generischen Typ-Parameter falls vorhanden müssen korrekt gesetzt werden

3.3 Übersetzung der Java-Konstrukte

Im Folgenden wird beschrieben, wie der Code-Importer die typischen Java-Konstrukte in Modellelemente übersetzt. Dabei werden alle Schritte zur Erstellung des plattformunabhängigen Kontrollflussgraphen zusammengefasst und die Ergebnisse der Übersetzung als

jABC4-Modelle dargestellt. Da Geschäftsprozesse Abfolge von Aktivitäten sind, werden diese im Kontrollfluss abgebildet. In Java wird die Sequenz von Anweisungen, die den Kontrollfluss festlegen, in Methoden realisiert. Daher reicht es nicht nur die Struktur einer Klasse in Modellen abzubilden. Mit Hilfe des Code-Importers kann der Kontrollfluss innerhalb einer Methoden zu einem Modellgraphen übersetzt werden.

Im weiteren wird also auf die Konstrukte eingegangen, die in einer Methodendeklaration vorkommen können. Für die Übersetzung bestimmter Konstrukte wurden SIBs aus der Basic-Bibliothek (s. Abschnitt 2.5.4) verwendet. Im Rahmen der Masterarbeit wurde eine neue Version der Basic-Bibliothek erstellt. Für diese wurden einige SIBs aus dem Plugin übernommen, einige SIBs wurden mithilfe des Code-Importers generiert und der Bibliothek hinzugefügt. Dafür wurde für jede Basic-Funktion eine Java-Methode implementiert und durch den Code-Importer in ein Modell überführt. Die Auflistung aller Basic-SIBs befindet sich im Anhang A.

3.3.1 Ausdrucksanweisungen

Zuweisung

Unter Zuweisung versteht man in Java den einfachsten Typ der Anweisungen, durch den eine Variable einen neuen Wert enthält. Einer Variable können drei verschiedene Arten von Werten zugewiesen werden: ein konstanter Wert, der Wert einer anderen Variable oder das Ergebnisses eines Ausdrucks, wie z.B. eines Methodenaufrufs. In diesem Abschnitt werden die ersten zwei Typen der Zuweisung betrachtet, diese werden auch als *einfache Zuweisungen* bezeichnet. Der dritte Fall wird im Rahmen anderer Ausdrucksarten behandelt. In jABC ist es möglich das Ergebnis einer Operation bzw. eines Methodenaufrufs einer Kontextvariable zuzuordnen. Diese Zuordnung entspricht der expliziten Zuweisung aus dem Quellcode.

Bei der Übersetzung werden die Kontextvariablen genau so benannt wie die Variablen im Quellcode. Im Falle einer einfachen Zuteilung des neuen Wertes, müssen lediglich die entsprechenden Kontextvariablen aktualisiert werden. Für die Aktualisierung einer Kontextvariable ist der Service-Graph *PutToContext* aus der *Basic*-Bibliothek zuständig. Dieser Graph nimmt als Eingabe einen Wert, welcher als Ergebnis zurückgegeben wird und so einer Kontextvariable zugewiesen werden kann. Der Eingabewert kann konstant sein oder aus einer existierenden Kontextvariable ausgelesen werden. Listing 3.2 zeigt zwei Beispiele für verschiedene Zuweisungen.

Listing 3.2: Einfache Zuweisungen

```
i = 5;
i = j;
```

Abbildung 3.9 zeigt die Verwendung von *PutToContext* für Beispiele aus Listing 3.2.

Variablendeklaration und Instanzerzeugung

Als Variablendeklaration bezeichnet man das Erstellen einer neuen Variable von einem bestimmten Typ. Diese enthält entweder einen initialen Wert oder wird mit `Null` initialisiert. Im Listing 3.3 werden drei Variablen deklariert. Die Variable i wird durch Zuweisung eines konstanten Wertes initialisiert, die Variablen *bigI* und *name* erhalten den Wert einer

Abbildung 3.9: Übersetzung der Zuweisungen i=5 (a) und i=j (b)

Klassen- bzw. einer Instanzvariable und die Instanz *pizza* wird durch einen Konstruktoraufruf initialisiert.

Listing 3.3: Verschiedene Variablendeklarationen

```
Integer i = 5;
BigInteger bigI = BigInteger.ZERO;
String name = pizza.name;
Pizza pizza = new Pizza(Size.Medium);
```

Für Variablendeklarationen durch Zuweisung eines Wertes wird als erstes eine neue Kontextvariable im Kontext des Modells angelegt. Der Name der neuen Kontextvariable entspricht dem Namen der Quellcode-Variable. Die Initialisierung wird durch den SIB *PutToContext* übersetzt. Auf diese Art wird die Deklaration der Variable *i* aus Listing 3.3 übersetzt.

Zugriffe auf Variablen einer Klasse oder eines Objektes wie `BigInteger.ZERO` bzw. `pizza.name` wird durch die Service-Graphen *GetStaticField* bzw. *GetInstanceField* übersetzt. Diese Graphen benutzen Java-Reflection um einen Wert einer Variable auszulesen. Der Graph *GetStaticField* wird verwendet, wenn es sich um statische Variablen (Klassenvariablen) handelt, wie in `BigInteger.ZERO`. Im Falle einer Instanzvariable, wie `pizza.name`, wird der Graph *GetInstanceField* benutzt. In der Tabelle 3.1 werden Ein- und Ausgabeparameter der beiden Graphen aufgelistet.

Service-Graph	**Eingabeparameter**	**Rückgabewert**
GetStaticField	`Class<?>` classWithField, `String` fieldName, `Class<T>` fieldType	T field
GetInstanceField	`Object` instanceWithField, `String` fieldName, `Class<T>` fieldType	T field

Tabelle 3.1: Spezifikation der Graphen *GetStaticField* und *GetInstanceField*

Bei einer Instanzerzeugung durch expliziten Konstruktor-Aufruf wird ein Service-Call auf den Konstruktor erstellt. Wie Methodenaufrufe können Konstruktoren Eingabeparameter besitzen. Diese können auf Konstanten, Variablen oder Aufzählungskonstanten gesetzt werden. Die Ausgabe des Service-Calls ist die neue Instanz, die der entsprechenden Kontextvariable zugewiesen wird. Abbildung 3.10 veranschaulicht die Übersetzung des Konstruktoraufrufs aus Listing 3.3.

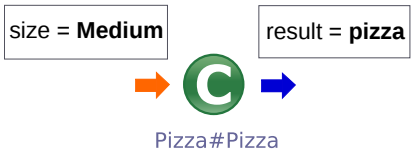


Abbildung 3.10: Übersetzung für Pizza pizza = new Pizza(Size.Medium)

Neue Variablen können auch als Ergebnisse eines Methodenaufrufes entstehen. Zu den Beispielen zählen

```
Pizza pizza = PizzaFactory.createPizza();
Pizza pizza = otherPizza.copy();
```

Diese Variablendeklarationen werden als Methodenaufrufe behandelt, deren Ergebnis einer neuen Kontextvariablen zugewiesen wird.

Methodenaufruf

Ein Methodenaufruf ist ein Ausdruck bestehend aus einem Methodennamen und Argumenten entsprechend der Methodendefinition. Da eine Methode immer einer Klasse oder einer Klasseninstanz zugeordnet ist, muss der Scope (Variblenname der Instanz oder Klassenname) ebenfalls angegeben werden. Ein Aufruf einer Methode wird typischerweise in einenService-Call oder ein Service-Graph übersetzt je nach Benutzereinstellungen. Listing 3.4 zeigt drei Bespiele für verschiedene Methodenaufrufe.

Listing 3.4: Beispiele für verschiedene Methodenaufrufe

```
pizza.cutInPieces(6);
PizzaFactory.createPizza(Size.SMALL);
deliverPizzas(pizzas);
```

Der Methodenaufruf `pizza.cutInPieces(6)` wird als ein Service-Call übersetzt. Dabei muss dem Service-Call eine Referenz auf die Instanz `pizza` übergeben werden. Service-Calls auf statische Methoden benötigen keine Instanzvariable. Für den Aufruf

```
PizzaFactory.createPizza(Size.SMALL);
```

reichen die Informationen `Class` und `Method`. Der dritte Methodenaufruf wird als Service-Graph dargestellt, dieser kann wie ein Service-Call Ein- und Ausgabeparameter besitzen. Die Ein- und Ausgabeparameter von Service-Calls bzw. Service-Graphen entsprechen der Methodendefinition. Falls in der Methode eine Instanzvariable verwendet wird, wird diese den Eingabeparametern hinzugefügt. Abbildung 3.11 zeigt die Ergebnisse der Übersetzung der Methodenaufrufe aus Listing 3.4.

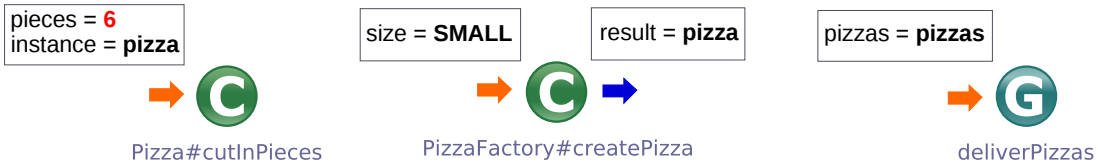


Abbildung 3.11: Übersetzung verschiedener Methodenaufrufe

Abbildung 3.12: Übersetzung des Post-Operators (a) und Pre-Operators (b)

Unäre Operatoren

Unäre Operatoren verändern den Wert einer Variable. Beispiele dafür sind Inkrementierung und Dekrementierung von primitiven Zahlen oder die Negation von booleschen Variablen:

```
i++;
i--;
!(true)
```

Für die unären Operatoren +, -, ++, - und ! wurden Service-Graphen erstellt und der *Basic*-Bibliothek hinzugefügt. Die Service-Graphen besitzen genau einen Ein- und einen Ausgabeparameter.

Man unterscheidet zwischen Post- und Pre-Operatoren. Der Unterschied wird vor allem bei Verwendung der Operatoren in Methodenaufrufen klar. Im Listing 3.5 wird die gleiche Methode mit verschiedenen Werten aufgerufen.

Listing 3.5: Pre- und Post-Operatoren

```
method(i++);
method(++i);
```

Die Übersetzung der beiden Methodenaufrufe wird in Abbildung 3.12 dargestellt.

Binäre Operatoren

Binäre Operatoren kommen in Ausdrücken vor und beziehen sich auf Booleans, Zahlen oder Strings. Die Operatoren werden je nach Datentyp unterschiedlich verwendet. Beispielsweise kann der Operator + für String-Konkatenation benutzt werden. Die strenge Typisierung in Java erlaubt es nicht, einen einen allgemeinen Graphen für jede Zahlenoperation zu erstellen. Die Oberklasse `Number` kann nicht als Ausgabe dienen, weil danach ein Cast auf einen Zahlentyp ausgeführt werden muss. Deswegen wurde die Basic-Bibliothek um Service-Graphen für jeden Datentyp erweitert (z.B. Addition mit Ausgabe `Double`, Addition mit Ausgabe `Integer`, String-Konkatenation usw.).

Falls der gesamte Ausdruck aus genau einer binären Operation besteht, wird dieser in einen Aufruf des entsprechenden Service-Graphen übersetzt. So benötigt man für den Ausdruck $i + j$ genau einen Aufruf des Graphen *Add_<Rückgabetyp>*. Die passenden Zahlentypen werden durch den Code-Importer automatisch ermittelt: Wenn i und j vom Typ Integer sind, wird der Graph *Add_Integer* ausgewählt. Wenn eine der Variable vom Typ Double ist, wird der Graph *Add_Double* verwendet. Wenn der Ausdruck eine komplexe Berechnung darstellt, wird jede Operation durch einen Graph-Aufruf dargestellt und die einzelnen SIBs werden entsprechend dem AST miteinander verbunden. Beispielsweise besteht der Übersetzungsgraph für den Ausdruck (i + 5) * (j - 2) aus drei SIBs, die in Abbildung 3.13 dargestellt werden. Der Additionsgraph `Add_Integer` hat die Eingabe-Parameter i und 5 und schreibt das Ergebnis in die Hilfsvariable `_i_plus_5_`. Der Substraktionsgraph `Substract_Integer` mit Parametern j und 2 schreibt das Ergebnis in die Variable `_j_minus_2_`. Im Multiplikationsgraphen `Multiply_Integer` werden die Ergebnisse der ersten beiden Operationen als Parameter verwendet.

Komplexe Bedingung mit mehreren booleschen Operatoren werden auf ähnliche Weise in eine Kette von Graph-Aufrufen transformiert. Die Zwischenergebnisse der Auswertung werden ebenfalls in Hilfsvariablen gespeichert. Im Unterschied zu Zahlenoperatoren endet die Auswertung einer Bedingung mit einem booleschen Wert, welcher durch die Branches „yes“ und „no“ abgebildet wird (s. Abbildung 3.14).

Für Konkatenation von Strings wird der Service-Graph *ConcatStrings* benutzt. Wenn einer der Operanden kein String ist, wird zuerst der Service-Call `toString` aufgerufen. Ein Beispiel wird in Abbildung 3.15 gezeigt.

Array-Operationen

Zu den Array-Operationen gehören Erzeugung, Zugriff und Veränderung eines Arrays. Für diese Operationen wurden Service-Graphen erstellt, die auf Methoden der Klasse `java.reflect.Array` basieren. Die Kurzschreibweise mit geschweiften Klammern für die Initialisierung eines Arrays wird in die Abfolge *NewArrayInstance* und mehrere *SetElement*-Graphen übersetzt.

Listing 3.6: Array-Initialisierung

```
String[] sArray= {"a", "b", "c"};
```

In Abbildung 3.16 wird das Ergebnis der Übersetzung einer Array-Initialisierung aus dem Listing 3.6 dargestellt. Der erste SIB des Modells erstellt eine leere Array-Instanz vom Typ `String[]`. Der zweite SIB weist dem ersten Element des Arrays den Wert „a“ zu. Die anderen SIBs verändern die restlichen Elemente im Array.

cast und instanceof

Durch den *Cast-Operator* kann der Datentyp eines Ausdrucks ausdrücklich festgelegt oder verändert werden. Ein Cast wird mit Hilfe des Service-Graphen *Cast* übersetzt. Dieser Graph benötigt als Parameter die Instanz, deren Typ verändert wird, und den neuen Typ. Das Ergebnis der Umwandlung wird in einer neuen Kontextvariable gespeichert. Für den Cast ohne explizite Variablenzuweisung wird eine neue Kontextvariable mit einem generierten Namen erzeugt. Der boolesche Operator `instanceof` wird durch Verwendung des Service-Graphen *IsInstance* übersetzt. Mit Hilfe des SIBs *IsInstance* kann überprüft werden, ob ein referenziertes Objekt (erstes Argument) zuweisungskompatibel zu einer Klasse ist, die als zweites Argument angegeben wird.

this, super und null

Wenn die Referenzen `this` oder `super` in einem Ausdruck vorkommen, wie z.B. bei der Referenz auf eine Instanzvariable (`this.name`) oder Methodenaufrufe (`super.methodInSuperClass()`), werden diese bei der Übersetzung berücksichtigt. Instanzvariablen werden genau so wie lokale Variablen durch Kontextvariablen repräsentiert und zu den Eingabevariablen des Modellgraphen hinzugefügt. In Methodenaufrufen werden `this` oder `super`-Referenzen benutzt, um die richtige Method zu finden. Referenzen `this` oder `super` in Methodenaufrufen werden bei der Suche nach dem richtigen `Method` benutzt. Die Rückgabe einer `this`-Referenz kann nicht übersetzt werden, denn Objekte mit mehreren Zuständen können nicht auf zustandslose Modellgraphen abgebildet werden.

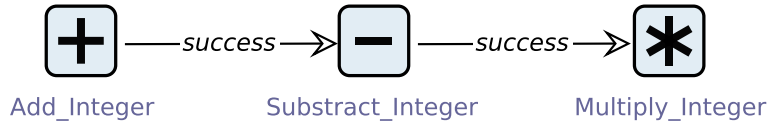


Abbildung 3.13: Übersetzung des binären Ausdrucks (i + 5) * (j - 2)

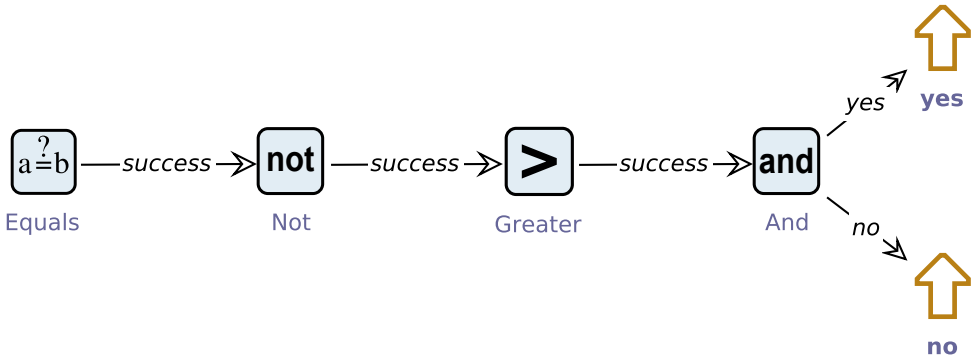


Abbildung 3.14: Übersetzung des booleschen Ausdrucks (i != null) && (i > 0)

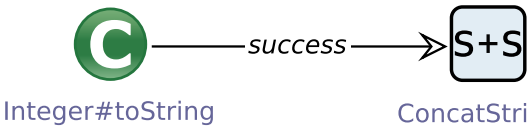


Abbildung 3.15: Übersetzung des Ausdrucks „Total pizza count: “ + size

Die Null-Referenz wird je nach Verwendung auf verschiedene Arten übersetzt. Für einen Vergleichsausdruck wie z.B. `a == null` wird der Service-Graph *EqualsNull* verwendet. Dieser übernimmt den Nullcheck für ein beliebiges Objekt. Wenn in eine Methode mit einem `Null`-Argument aufgerufen wird, wird für dieses Argument eine generierte uninitialisierte Kontextvariable verwendet. Da diese Kontextvariable nicht weiter verwendet wird, wird die Anzahl der Modell-Variablen unnötig erhöht. Die Verwendung von Null-Argumenten ist daher abzuraten und sollte vor Beginn der Modellbildung beseitigt werden.

Im Falle einer Variableninitialisierung mit `Null` wird einfach eine neue Variable dem Modellkontext hinzugefügt. Diese zeigt automatisch ohne explizite Initialisierung auf die Null-Referenz. Ähnlich wird die Anweisung

```
return null;
```

behandelt. Zurückgegeben wird einfach eine neue uninitialisierte Instanz des Rückgabetyps der Methode.

Verschachtelte Ausdrücke

Ein verschachtelter Ausdruck besteht aus mehreren Ausdrücken, die hierarchisch geordnet sind. Beispielsweise können Argumente bzw. die Instanz in einem Methodenaufruf durch weitere Methodenaufrufe dargestellt werden, wie Listing 3.7 zeigt.

Listing 3.7: Verschachtelte Ausdrücke in Methodenaufrufen

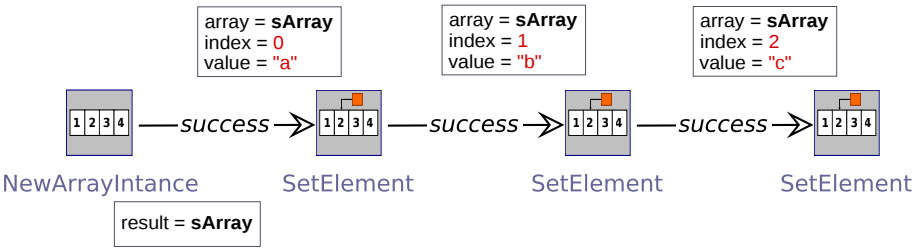


Abbildung 3.16: Übersetzung der Array-Initialisierung

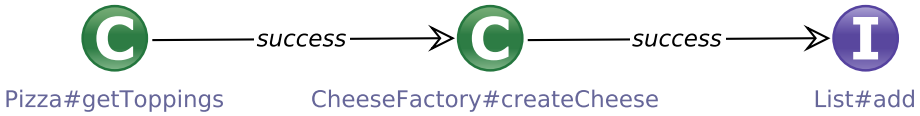


Abbildung 3.17: Übersetzung eines verschachtelten Ausdrucks

```
pizza.getToppings().add(CheeseFactory.createCheese());
```

Da die Reihenfolge der Aufrufe immer eindeutig ist, können solche Ausdrücke ohne Probleme geparst werden. Im Beispiel kann die Methode `add` erst dann aufgerufen werden, wenn die Ergebnisse der Aufrufe `pizza.getToppings()` und `CheeseFactory.getCheese()` feststehen. Diese Ergebnisse werden in generierte Kontextvariablen geschrieben. Dann kann der Service-Call für `add` mit der Instanzvariable `_toppings_` und dem Parameter `_cheese_` ausgeführt werden. Das Ergebnis der Übersetzung des Ausdrucks aus Listing 3.7 wird in Abbildung 3.17 dargestellt.

3.3.2 Blockanweisungen

Eine *Blockanweisung* fasst eine Gruppe von Anweisungen (engl. *statement*) zusammen, die nacheinander ausgeführt werden. Die Anweisungen eines Blocks werden durch geschweifte Klammern eingegrenzt:

Listing 3.8: Blockanweisung

```
{
  Anweisung_1;
  Anweisung_2;
  ...
}
```

Blöcke werden oft im Zusammenhang mit bedingten Anweisungen, Schleifen und Ausnahmeverarbeitung eingesetzt. Außerdem wird der Rumpf einer Methodendefinition als Block formuliert. Bei der Übersetzung in einen Graphen wird jede Anweisung eines Blocks durch einen Knoten oder einen Teilgraphen dargestellt und diese werden miteinander durch Kanten verbunden (s. Abbildung 3.18)

3.3.3 Bedingte Anweisungen

If-Anweisungen

`If`-Anweisungen bestehen aus einer Bedingung und einer Blockanweisung. Die Anweisungen im `Then`-Block werden nur dann ausgeführt, wenn die Bedingung erfüllt ist. Optional ist ein `Else`-Block, der nur ausgeführt wird, wenn die Bedingung nicht erfüllt ist.

Wie eine bedingte Anweisung in einen Modellgraphen übersetzt wird, kann der Abbildung 3.19 entnommen werden. Als Erstes wird die Bedingung in ein oder mehrere SIBs transformiert. Der letzte SIB der übersetzen Bedingung endet mit Branches „yes“ und „no“. Falls die Bedingung erfüllt ist, wird der Branch „yes“ gewählt und der `Then`-Block wird ausgeführt. Wenn der Branch

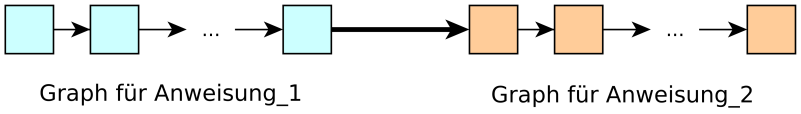


Abbildung 3.18: Übersetzung einer Blockanweisung

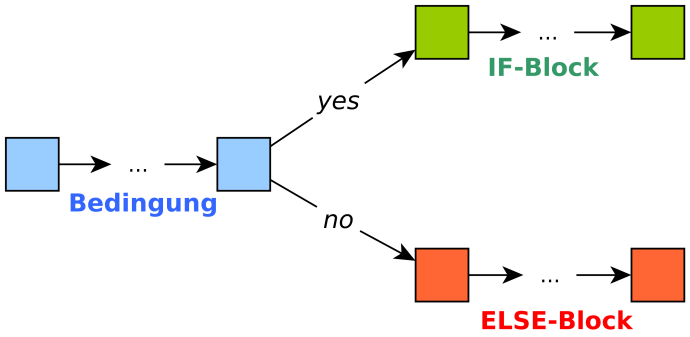


Abbildung 3.19: Übersetzung einer If-Else-Anweisung

„no“ verfolgt wird, kommt der `Else`-Block zum Einsatz.

Müssen mehrere Bedingungen überprüft werden, benutzt man häufig verkettete Anweisungen mit `else if`. Bei der Übersetzung werden die einzelnen Bedingungen nacheinander überprüft. Wenn nach der ersten Bedingung der Branch „no“ gewählt wird, wird die zweite Bedingung überprüft usw.

Es existieren mehrere Möglichkeiten, eine Bedingung in Java auszudrücken. Im einfachsten Fall wird eine boolesche Variable ausgewertet oder das Ergebnis einer booleschen Methode. Etwas schwer zu verstehen sind Bedingungen, die durch einen booleschen Ausdruck dargestellt werden. Folgende Beispiele zeigen, wie verschiedene Bedingungen bei der Modellbildung behandelt werden:

Listing 3.9: Verschiedene Bedingungen einer If-Anweisung

```
(a) if (pizzaIsReady){
    deliverPizza();
}
(b) if (pizza.hasSauce()){
    pizza.addToppings();
}
(c) if (totalPrice > 20.00){
    addFreeCola();
}
```

Bedingungsoperator

Wenn abhängig von einer Bedingung verschiedene Werte ausgewählt werden, wird häufig der Bedingungsoperator `?` verwendet. Beispielsweise kann einer Variable je nach Auswertungsergebnis ein unterschiedlicher Wert zugewiesen werden:

Listing 3.10: Variableninitialisierung mit Bedingungsoperator

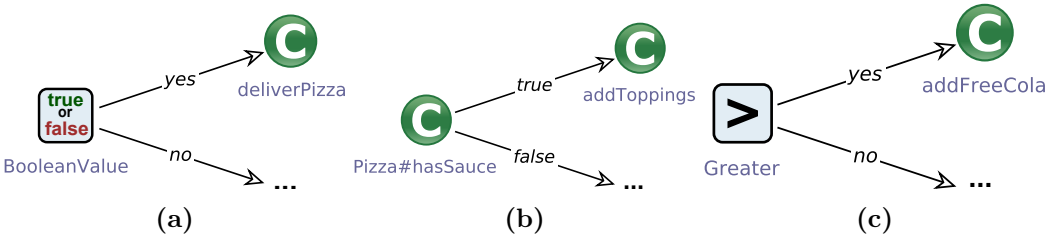


Abbildung 3.20: Übersetzung verschiedener Bedingungen aus Listing 3.9

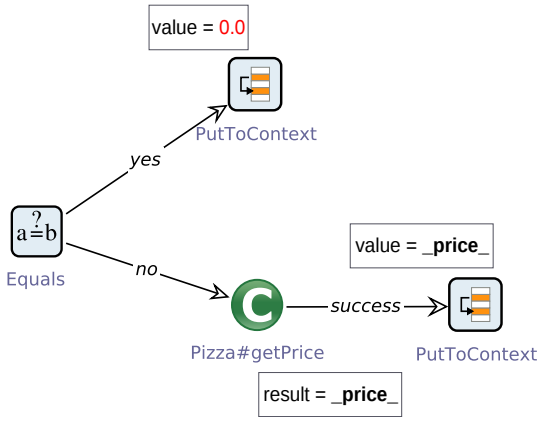


Abbildung 3.21: Der Modellgraph für das Beispiel aus Listing 3.10

```
Double price = (pizza == null) ? 0.0 : pizza.getPrice();
```

Der Bedingungsoperator wird ähnlich wie eine `If-Else`-Anweisung in einen Graphen übersetzt. Abbildung 3.21 zeigt die Übersetzung des Ausdrucks aus Listing 3.10.

Switch-Anweisung

Eine `Switch`-Anweisung stellt eine Spezialform von Fallunterscheidungen dar. Im `Switch`-Block gibt es eine Reihe von unterschiedlichen Sprungzielen, die mit `case` markiert sind. Als Bedingungen sind Aufzählungen, Ganzzahlen und Strings (seit Java 7) erlaubt. In jABC 4 enthalten Service-Calls auf Methoden, die ein Aufzählungselement zurückgeben, die den Aufzählungselementen entsprechenden Ausgabe-Branches. Beispielsweise hat der Service-Call für `pizza.getCheese()` die Branches *Mozarella*, *Gouda* und *Massdamer* (s. Listing 3.11).

Listing 3.11: Switch mit Enumeration

```
switch(pizza.getCheese()){
 case Mozarella: cheese = new Mozarella(); break;
 case Gouda: cheese = new Gouda(); break;
 case Maasdamer: cheese = new Maasdamer(); break;
}
```

Diese Eigenschaft lässt sich bei der Übersetzung der `Switch`-Anweisung gut ausnutzen: Abbildung 3.22 zeigt den Modellgraphen für die Anweisung aus Listing 3.11.

Der Code aus Listing 3.11 kann auch mithilfe der Strings implementiert werden:

Listing 3.12: Switch mit Vergleich

```
switch(pizza.getCheese()){
```

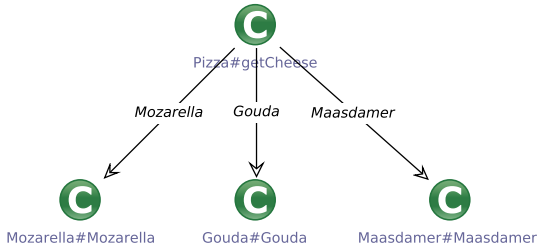


Abbildung 3.22: Übersetzung einer Switch-Anweisung mit Aufzählungskonstanten

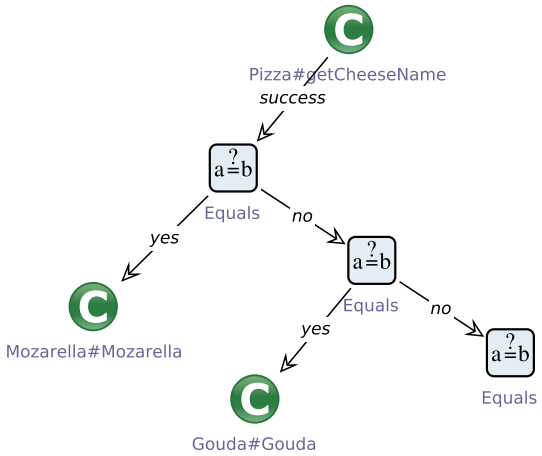


Abbildung 3.23: SLG für Switch mit Strings

```
  case "mozarella": cheese = new Mozarella(); break;
  case "gouda": cheese = new Gouda(); break;
  ...
}
```

Bei der Übersetzung müssen hier einzelne Strings miteinander verglichen werden. Das Ergebnis des Service-Calls `pizza.getCheese()` wird zuerst mit dem String „mozarella“ verglichen, danach mit „gouda“ usw. Erst wenn ein Vergleich erfolgreich ist, werden der Teilgraph des entsprechenden `Case`-Blocks ausgeführt.

Eine besondere Gefahr stellt das sogenannte *Fall-Through* dar. Wenn ein `Case`-Block nicht mit einer `Break` oder `Return`-Anweisung endet, muss der nächste `Case`-Block ausgeführt werden. Im Beispiel aus Listing 3.13 wird die Backzeit einer großen Pizza um 15 Minuten erhöht, für eine mittlere Pizza beträgt die Backzeit 10 Minuten und für eine kleine Pizza lediglich fünf.

Listing 3.13: Fall-Through

```
Integer bakingTime = 0;
switch(pizza.getSize()){
  case LARGE: bakingTime = increaseBakingTime(bakingTime, 5);
  case MEDIUM: bakingTime = increaseBakingTime(bakingTime, 5);
  case SMALL: bakingTime = increaseBakingTime(bakingTime, 5);
}
```

Die Übersetzung dazu berücksichtigt den Fall-Through indem die Knoten *increaseBakingTime* miteinander verbunden werden. Dies wird in Abbildung 3.24 veranschaulicht.

Ein weiteres Problem stellen die `Default`-Blöcke dar. Die Sprungmarke `default` deckt alle Fälle ab, die nicht mit anderen `Case`-Blöcken übereinstimmen. Beispielsweise werden durch `default` alle Pizzengrößen abgedeckt, die nicht *LARGE* sind:

Listing 3.14: Switch mit Default

```
switch(pizza.getSize()){
  case LARGE: addFreeCola();
  default: deliverPizza();
}
```

Im Modellgraphen werden die im `default` versteckten Konstanten explizit als Kanten dargestellt. Diese Kanten führen zu dem Knoten *deliverPizza*, denn das ist die Übersetzung des `Default`-Blocks (vgl. Abbildung 3.25).

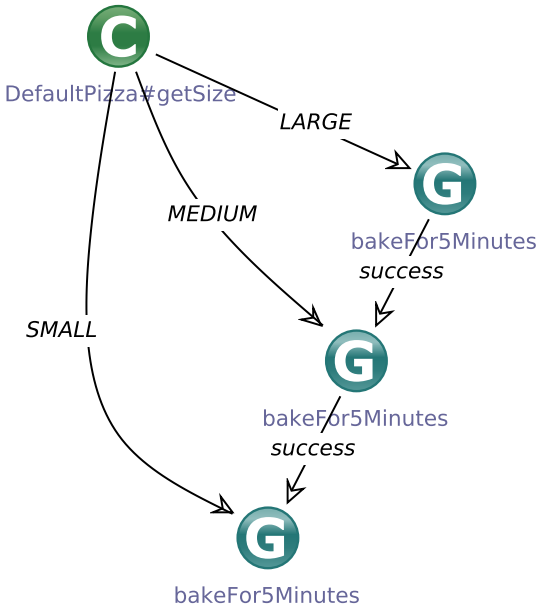


Abbildung 3.24: SLG für Switch mit Fall-Through

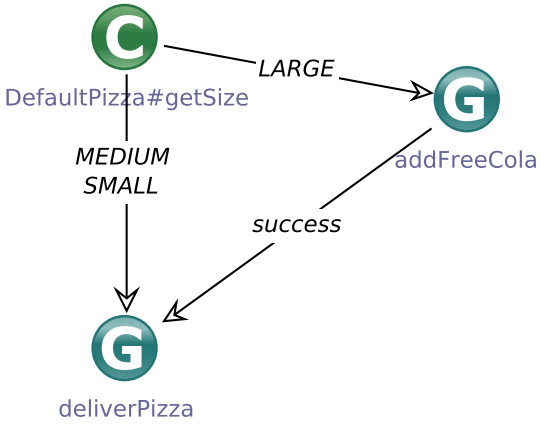


Abbildung 3.25: SLG für Switch mit Default

Fall es keinen expliziten `Default`-Block gibt und nicht alle Aufzählungskonstanten durch Cases abgedeckt sind, werden die Kanten der fehlenden Konstanten zur ersten Anweisung nach dem `Switch`-Block eingefügt. Listing 3.15 zeigt eine `Switch`-Anweisung ohne Sprungziel für die Aufzählungskonstante `SMALL`.

Listing 3.15: Switch ohne Default

```
switch(pizza.getSize()){
  case LARGE:
  case MEDIUM:  addFreeCola();
}
deliverPizza();
```

In Abbildung 3.26 wird der generierte Modellgraph zum Listing 3.15 dargestellt.

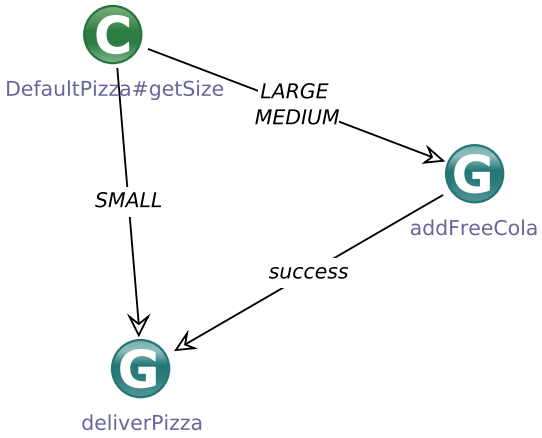


Abbildung 3.26: SLG für Switch ohne Default

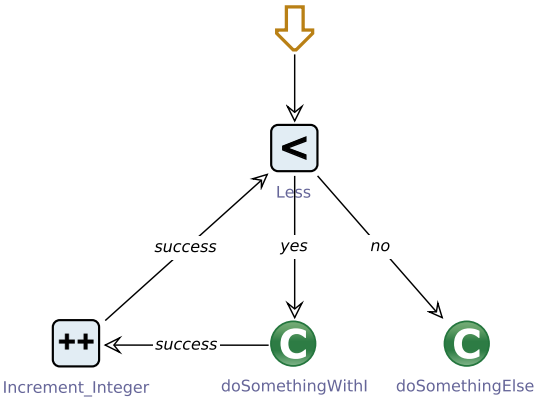


Abbildung 3.27: Modellgraph für die While-Schleife aus Listing 3.16

3.3.4 Iterationsanweisungen

While-Schleife

Die `While`-Schleife ist eine abweisende Schleife, die vor jedem Schleifeneintritt die Schleifenbedingung prüft. Ist die Bedingung wahr, wird der Rumpf-Block ausgeführt, andernfalls endet die Schleife. Ein Bespiel für eine `While`-Schleife wird in Listing 3.16 dargestellt.

Listing 3.16: While-Schleife

```
while(i < 5){
  doSomethingWithI(i);
  i++;
}
doSomethingElse();
```

Der Modellgraph für eine `While`-Schleife ähnelt dem Graphen für eine `If`-Anweisung. Die Bedingung wird in eine Kette von SIBs übersetzt, die mit einem positiven und einem negativen Branch endet. Nach dem positiven Branch kommt der Teilgraph des Schleifenrumpfs, dessen letzte Kante zur Bedingung führt. Der negative Branch bedeutet das Ende der Schleifenausführung und zeigt auf den Knoten der ersten Anweisung nach der Schleife. Der für Listing 3.16 korrespondierende Modellgraph wird in Abbildung 3.27 veranschaulicht.

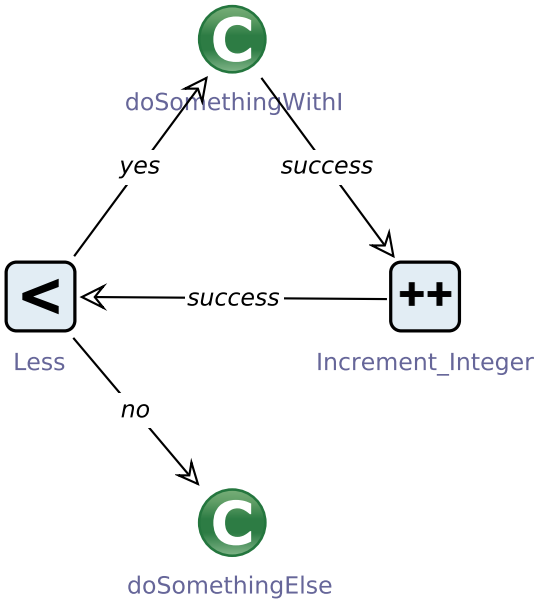


Abbildung 3.28: Modellgraph für die Do-While-Schleife aus Listing 3.17

Do-While Schleife

Die `Do-While`-Schleife ist eine annehmende Schleife, da die Schleifenbedingung erst nach dem Schleifendurchgang überprüft wird. Der Rumpf-Block wird also mindestens ein Mal ausgeführt. Bei der Übersetzung zunächst der Rumpf in einen Teilgraphen übersetzt. Der letzte Knoten des Rumpfs wird mit dem ersten Knoten der Schleifenbedingung verbunden. Listing 3.17 und Abbildung 3.28 zeigen eine Beispiel-Schleife und den resutierenden Modellgraphen.

Listing 3.17: Do-While-Schleife

```
do{
    doSomethingWithI(i);
    i++;
}
while(i < 5);
doSomethingElse();
```

For-Schleife

Der Kopf der `For`-Schleife besteht aus drei Teilen, wobei jeder für sich optional ist. Im ersten Teil werden Initialisierungsausdrücke aufgelistet, diese definieren und initialisieren die Zählervariablen. Im Testteil wird der Bedingungsausdruck für die Fortsertung der Schleife aufgeführt. Der letzte Teil besteht aus Update-Ausdrücken, hier werden die Schleifenzähler verändert. Der Update-Teil wird nach jedem Durchlauf der Schleife ausgeführt, bevor der Testausdruck das nächste Mal ausgewertet wird.

Die Übersetzung einer `For`-Schleife in einen SLG ergibt folgende Graphenstruktur. Die ersten Knoten des Graphen beschreiben den Initialisierungsteil, gefolgt von den Knoten, die die Testbedingung darstellen. Der letzte Knoten des Bedingungsgraphen enthält einen positiven und einen negativen Branch. Der positive Branch führt zum ersten SIB der Blockanweisung innerhalb der Schleife und der negative Branch führt zur ersten Anweisung nach der Schleife. Nach dem letzten SIB der Blockanweisung kommen die Update-SIBs, der mit dem ersten Testbedingungsknoten verbunden werden. Listing 3.18 zeigt eine `For`-Schleife, die in Abbildung 3.29 übersetzt wird.

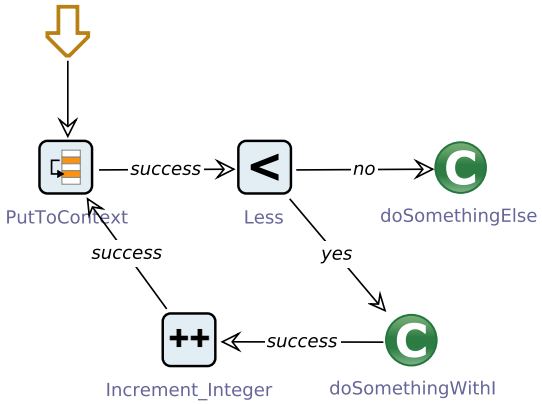


Abbildung 3.29: SLG für die For-Schleife aus Listing 3.18

Listing 3.18: For-Schleife

```
for(int i=0, i<5; i++){
    doSomethingWithI(i);
}
doSomethingElse();
```

Foreach-Schleife

Einen Sonderfall einer `For`-Schleife stellt die sogenannte `Foreach`-Schleife dar. Diese wird benutzt, wenn über eine Kollektion von Objekten iteriert wird. Der Kopf der `Foreach`-Schleife besteht aus drei Teilen: dem Typ und Variablennamen eines Elements und dem Namen der Kollektion (s. Listing 3.19).

Listing 3.19: For-Schleife

```
for(Pizza pizza : pizzas){
    bakePizza(pizza);
    cutPizza(pizza);
    putToBox(pizza);
}
deliverPizzas(pizzas);
```

Bei der Übersetzung einer `Foreach`-Schleife wird der Service-Graph *Iterate* aus den Basic-SIBs verwendet. Falls eine Kollektion nicht leer ist, gibt der *Iterate*-Graph das nächste Element im Branch *next* aus. Am Ende der Iteration wird der Branch *exit* ausgewählt. Der Modellgraph für das Beispiel aus Listing 3.19 wird in Abbildung 3.30 dargestellt.

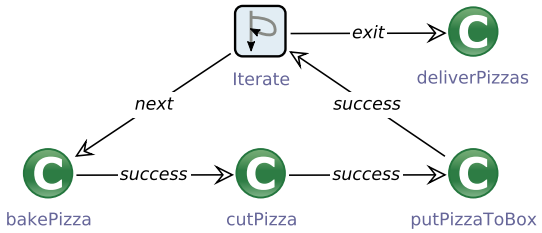


Abbildung 3.30: SLG für die Foreach-Schleife aus Listing 3.19

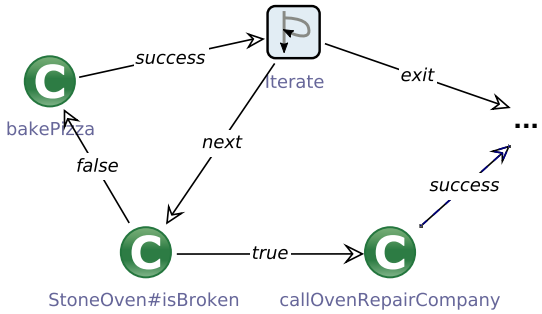


Abbildung 3.31: Modellgraph für die Foreach-Schleife mit einem Break

3.3.5 Sprunganweisungen

Break

Manchmal lässt sich im Java-Code das Schlüsselwort `break` finden. Wird innerhalb einer Schleife eine `Break`-Anweisung eingesetzt, so wird der Schleifendurchlauf beendet und die Abarbeitung bei der ersten Anweisung nach der Schleife fortgeführt. Beispielsweise sollen keine Pizzen mehr gebacken werden, wenn der Ofen nicht betriebsfähig ist:

Listing 3.20: Break in einer For-Schleife

```
for(Pizza pizza : pizzas){
  if (stoneOven.isBroken()){
        callOvenRepairCompany();
        break;
  }
  bakePizza(pizza);
}
deliverPizzas(pizzas);
```

Bei der Modellbildung wird eine `Break`-Anweisung als Kante (oder Kanten) zum ersten Knoten außerhalb der Schleife hinzugefügt. Abbildung 3.31 zeigt den Modellgraphen zum Quellcode aus Listing 3.20.

Continue

Innerhalb einer Schleife lässt sich eine `Continue`-Anweisung einsetzen, die nicht wie `Break` die Schleife beendet, sondern zum Schleifenkopf zurückgeht. Und zwar wird nach dem `continue` die Testanweisung überprüft und die Schleife wird entweder weiter durchlaufen oder die Ausführung der Schleife endet. Listing 3.21 benutzt `continue`, um das Backen einer nicht existierenden Pizza zu verhindern:

Listing 3.21: Continue in einer For-Schleife

```
for(Pizza pizza : pizzas){
  if (pizza.isToGo()){
      continue;
  }
  putPizzaOnPlate(pizza);
}
...
```

Im Modellgraphen aus Abbildung 3.32 lässt sich die `Continue`-Anweisung als „yes“-Kante führend zu *Iterate* finden.

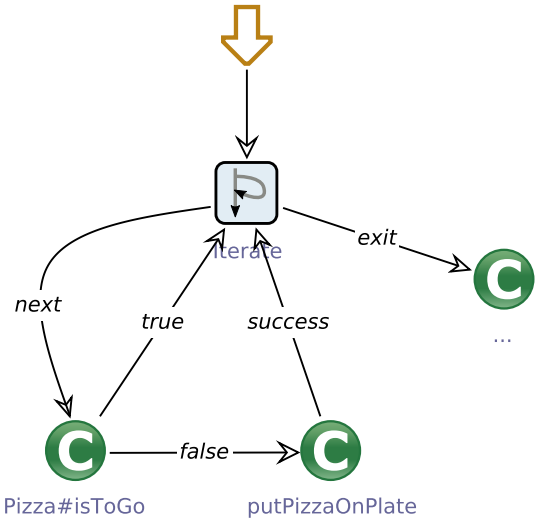


Abbildung 3.32: Modellgraph für die Foreach-Schleife mit einem Continue

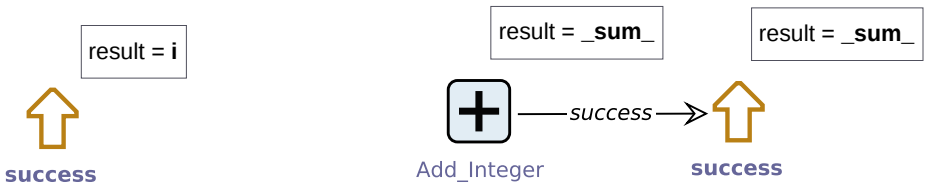
Return

Am Ende einer Methode mit Rückgabewert findet man eine `Return`-Anweisung. Nach dem Stichwort `return` kann entweder eine Variable oder ein Ausdruck stehen. Der Typ der Variable oder des Ergebnis des Ausdrucks stimmt mit dem Rückgabewert der Methode überein oder kann davon abgeleitet werden. Jede `Return`-Anweisung wird in einen Modell-Branch *success* übersetzt. Wenn die Anweisung einen auswertbaren Ausdruck enthält (z.B. return i + 5;) wird dieser zunächst in einen Teilgraphen überführt. Das Ergebnis der letzten Operation bzw. des letzten Methodenaufrufes wird dem Modell-Branch hinzugefügt. Abbildung 3.33 veranschaulicht einen einfachen und einen komplexeren Fall.

Bei mehreren `Return`-Anweisungen innerhalb einer Methode entsteht ein Problem, denn ein SLG kann nicht mehrere Modell-Branches mit dem gleichen Namen besitzen. Deswegen werden die Ergebnisse aller `Return`-Anweisungen einer neuen Variable zugewiesen und die Variable wird dem Modell-Branch hinzugefügt. Abbildung 3.34 veranschaulicht den Modellgraphen für der Quellcode aus Listing 3.22.

Listing 3.22: Eine Methode mit zwei Return-Anweisungen

```
if (...){
    return a;
}
else{
    return b;
}
```



(a) return i **(b)** return i + 5

Abbildung 3.33: Modellgraphen für Return-Anweisungen

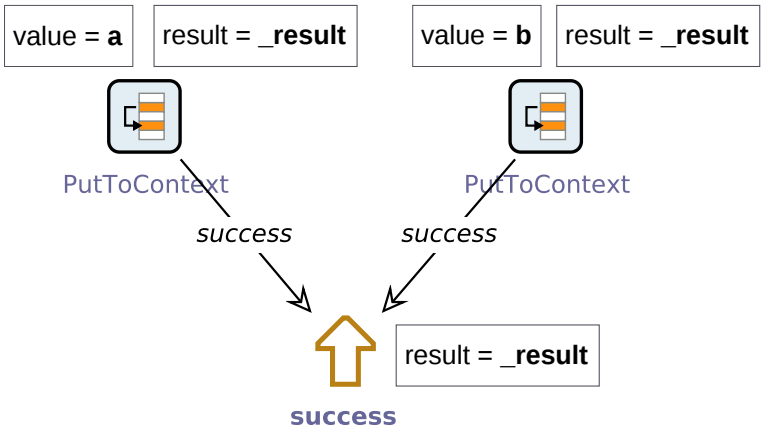


Abbildung 3.34: SLG für Return-Anweisungen aus Listing 3.22

Throw

Die Throw-Anweisung wird ebenfalls in einen Modell-Branch übersetzt. Der Name des Branches entspricht der Klasse der Ausnahme. Die entsprechende Exception wird als Ausgabe-Variable im Branch gespeichert. Beispielsweise wird eine Exception geworfen, wenn eine Pizza nicht existiert:

Listing 3.23: Eine Throw-Anweisung

```
if (pizza == null){
   NoPizzaException e = new NoPizzaException();
   throw e;
}
...
```

Der Modellgraph zu Listing 3.23 wird in Abbildung 3.35 dargestellt.

3.3.6 Ausnahmebehandlung

Ausnahmen in Java können entweder mit `Try-Catch`-Anweisungen behandelt oder an die aufrufende Methode weitergereicht werden. Wird eine Exception weitergeleitet, so ist diese in der Methodendeklaration nach dem Schlüsselwort `throws` aufgeführt. Diese Ausnahmen müssen auch im Modell weitergegeben werden. Dazu wird der Exception-Branch erstellt, welcher nach Auftritt einer Ausnahme ausgegeben wird. Beispielsweise kann die Methode `methodNotHandlingExceptions` aus Listing 3.24 keine Ausnahmen behandeln und leitet diese einfach weiter.

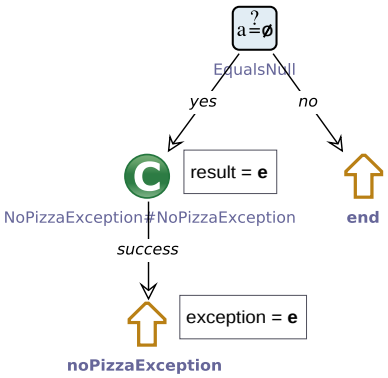


Abbildung 3.35: SLG für Throw-Anweisungen aus Listing 3.23

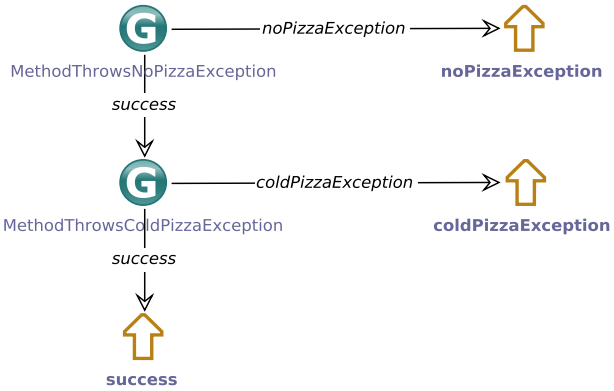


Abbildung 3.36: Modellgraph für methodNotHandlingExceptions

Listing 3.24: Weiterleiten einer Exception

```
public void methodNotHandlingExceptions() throws NoPizzaException,
            ColdPizzaException{

  methodThrowsNoPizzaException();
  methodThrowsColdPizzaException();

}
```

Bei der Übersetzung werden dem Modellgraphen die Ausgaben *noPizzaException* und *coldPizzaException* hinzugefügt. Die Service-Blocks, die eine Exception werfen, erhalten eine Kante zu dem entsprechenden Modell-Branch mit der Exception. Abbildung 3.36 zeigt den Modellgraphen für die Beispielmethode aus Listing 3.24.

Wenn eine Exception behandelt wird, findet die Behandlung in einem `Catch`-Block statt, welcher nach einem `Try`-Block auftritt. Nach der Behandlung wird der optionale `Finally`-Block ausgeführt. Dieser wird ebenfalls ausgeführt, wenn im `Try`-Block keine Exception auftritt. Listing 3.25 zeigt eine Methode, die Pizza-Exceptions behandelt.

Listing 3.25: Behandeln einer Exception

```
public void methodHandlesExceptions(){
  try{
    methodThrowsNoPizzaException();
    methodThrowsColdPizzaException();
  }
  catch(NoPizzaException e){
    prepareNewPizza();
  }
  catch(ColdPizzaException e){
    warmUpPizza();
  }
  finally{
    finish();
  }
}
```

Bei der Übersetzung wird der `Try`-Block wie eine normale Blockanweisung behandelt. Ein `Catch`-Block wird nur dann ausgeführt, wenn die Ausnahme auftritt, d.h. wenn der entsprechende Branch ausgewählt wird. In Abbildung 3.25 wird der Modellgraph zur Methode aus Listing 3.25 gezeigt. Zu beachten ist, dass der `Catch`-Block für die `NoPizza`-Ausnahme nur dann ausgeführt wird, wenn das Ergebnis des Service-Calls *methodThrowsNoPizzaException* die entsprechende Ausnahme ist. Das gleiche gilt für die Ausnahme `ColdPizzaException`.

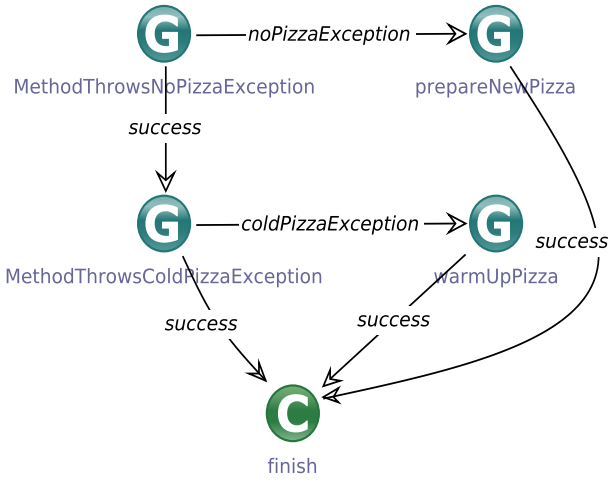


Abbildung 3.37: Modellgraph für methodHandlesExceptions

3.4 Einschränkungen und Voraussetzungen

Im Rahmen dieser Masterarbeit wurden nicht alle Java-Konstrukte berücksichtigt. Die zu Problemen führenden Konstrukte sollten vor der Anwendung des Code-Importers durch Refactorings beseitigt werden. Die automatische Übersetzung des Quellcodes in Modelle ist nur dann möglich, wenn alle folgende Voraussetzungen erfüllt werden:

- Der Quellcode beinhaltet keine syntaktischen Fehler, d.h. der Programmcode ist kompilierbar.
- Alle zu übersetzenden Java-Klassen stammen aus dem gleichen Zeichensatz.
- Die zu übersetzende Klasse ist kein Interface und deren Methoden sind nicht abstrakt und referenzieren keine abstrakten Methoden, deren Implementierung unbekannt ist.
- Pro Quelldatei erfolgt genau eine Klassendefinition. Es werden keine anonymen Klassen deklariert, wie zum Beispiel der `Comparator` in Listing 3.26.

Listing 3.26: Deklaration einer anonymen Implementierung

```
Collections.sort(collection, new CollectionElementComparator{

    @Override
    public int compare(CollectionElement e1, CollectionElement e2) {
        ...
    }

});
```

- Aufzählungen enthalten Elemente, die finalen statischen Charakter besitzen. Alle Aufzählungen werden in separaten Dateien deklariert.
- Es werden keine Konstruktoren der aktuellen Klasse durch die Verwendung von `this` aufgerufen.
- Keine der zu übersetzenden Methoden gibt `this` zurück.
- Statische Initialisierungsblöcke werden bei der Übersetzung nicht berücksichtigt, diese sollen vorher als Methoden umgesetzt werden.

- Methoden erhalten keine Annotationen, außer `@Override`. Andere Annotationen werden nicht berücksichtigt.
- Der zu übersetzende Programmcode verwendet nicht die Schlüsselwörter `native`, `synchronized`, `strictfp`, `transient` und `volatile`. Außerdem treten die Sprunganweisungen `break` und `continue` nur ohne Spezifikation eines Rücksprungpunktes auf.
- In den Methodendeklarationen wird die Abkürzungsnotation `...` nicht verwendet. Diese kann durch ein echtes Array bzw. eine Liste ersetzt werden (s. Listing 3.27).

Listing 3.27: Refactoring einer Methodendeklaration mit ...

```
public void method(String... messages){ // kann nicht übersetzt werden
    ...
}

public void method(List<String> messages){ // kann übersetzt werden
    ...
}
```

Sollte eine Methodendeklaration eine der Anforderung nicht erfüllen, wird der Übersetzungsvorgang sofort mit einer entsprechenden Fehlermeldung abgebrochen. Folgende Java-Konstruktionen verursachen zwar keinen Ausnahmezustand, sollten im Allgemeinen jedoch vermieden werden, da diese auf Modellebene zu Problemen führen können. Es wird empfohlen, diese Konstrukte vor der Modellbildung anzupassen.

- Rückgabe von `Null`-Referenzen kann Nullpointer-Exceptions bei der Ausführung der Modelle verursachen, wenn auf diese im weiteren Verlauf zugegriffen wird. Diese Referenzen sollten vor der Modellbildung umgeschrieben werden. Beispielsweise kann eine leere Instanz des Objektes zurückgegeben werden. Natürlich hängt die Behandlung der `Null`-Referenzen von der Geschäftslogik ab. Es muss daher für jede betroffene Methode überprüft werden, ob die Rückgabe eines leeren Objektes sinnvoll ist.
- Eingabe von `Null`-Objekten als Methodenparameter kann durch *Overloading* vermieden werden. Overloading wird in Listing 3.28 erläutert.

Listing 3.28: Overloading in Java

```
public void method(String s, Integer i){
    ...
}

public void method(String s){ // Default-Wert für i ist 0
    method(s, 0);
}
```

- Arrays zu technischen Details und sind nicht erweiterbar. Modelle, die Arrays verwenden, sind unflexibel. Die Verwendung von Listen oder anderen Kollektionen ist für die Modellierung besser geeignet.

3.5 Korrektheitsüberprüfung

In diesem Abschnitt wird diskutiert, wann die Übersetzung des Quellcodes in ein Modell als *korrekt* bezeichnet werden kann. Die Übersetzung kann auf verschiedene Korrekheitskriterien überprüft werden. Beispielsweise erfordert die *syntaktische* Korrekheit lediglich, dass das erzeugte Modell wohlgeformt ist. Unter *semantischen* Korrektheit wird die Erhaltung der Bedeutung des Codes verstanden. Das heißt, bei der Modellbildung soll das Verhalten des Programms nicht verändert

werden. Während die syntaktischen Eigenschaften mithilfe des jABC-Editors leicht zu überprüfen sind, kann man die Semantik nur schwer verifizieren. Es existieren Ansätze zu automatischen Verifikation von Modellen, die sogenannten *Model-Checker*. Model-Checker sind Programme, die bei Eingabe System-Modell M und Spezifikation S das Ergebnis der Überprüfung, ob $L(M)$ äquivalent zu $L(S)$ ist, ausgeben. Die Spezifikation wird oft als endlicher Automat (oder in äquivalenter Form) dargestellt. Für die Problemstellung der Äquivalenz aller möglichen Java-Konstrukte[1] und deren Übersetzungen sind Model-Checking-Verfahren nicht anwendbar, denn es existieren unendlich viele Java-Programme.

Im Rahmen dieser Masterarbeit wurde die Korrektheitsüberprüfung auf einzelne Konstrukte reduziert. Mithilfe der strukturellen Induktion wird im Folgenden gezeigt, dass die Korrektheitsüberprüfung auf eine bestimmte Untermenge der möglichen Java-Ausdrücke reduziert werden kann. Im ersten Schritt wird gezeigt, dass eine beliebige Ausdrucksanweisung korrekt übersetzt wird. Im nächsten Schritt kann gezeigt werden, dass alle anderen Anweisungsarten ebenfalls korrekt übersetzt werden.

Ausdrücke

Die Menge aller möglichen Ausdrucksanweisungen im Java-Code kann als induktive Menge über folgende atomare Elemente definiert werden:

- Variablen und Literale (Zahlen und Strings)
- Instanz- oder Klassenvariablen (z.B. `array.length`)
- Ausdrücke mit genau einem unären oder binären Java-Operator mit atomaren Argumenten (z.B. i+1)
- Methodenaufrufe mit atomarem Scope und atomaren Argumenten
- Array-Ausdrücke mit atomarem Array-Namen und atomaren Argumenten
- Wort-Operatoren wie z.B. `cast` und `instanceof` mit atomaren Argumenten

Diese Ausdrücke können als atomar betrachtet werden, weil jeder Ausdruck durch genau einen SIB übersetzt wird. Beispielsweise wird der Ausdruck i+1 durch den SIB *Add* übersetzt, `array.length` wird als der SIB *GetInstanceField* im Modellgraphen dargestellt. Ob die einzelne SIBs das korrekte Verhalten aufweisen, kann durch Tests mit verschiedenen Eingaben überprüft werden.

Durch Verschachtelung entstehen unendlich viele Ausdrücke, deren Übersetzung auf Korrekheit überprüft werden muss. Allerdings kann man durch strukturelle Induktion zeigen, dass dies nicht nötig ist. Lässt sich die Übersetzungskorrektheit für jeden atomaren Ausdruck durch Tests beweisen (Induktionsanfang), dann lässt sich daraus die Annahme ableiten, dass alle atomare Ausdrücke innerhalb des verschachtelten Ausdrucks korrekt übersetzt werden (Induktionsschluss). Für den Konstruktor *Verschachtelung* muss gezeigt werden, dass aus dem Induktionsschluss folgt, der gesamte verschachtelte Ausdruck wird korrekt übersetzt. Jeder verschachtelte Ausdruck ist als ein eindeutiger binärer Auswertungsbaum darstellbar. Da bei der Übersetzung die Reihenfolge der erzeugten SIBs dem Auswertungsbaum entspricht, kann davon ausgegangen werden, dass das Gesamtergebnis korrekt berechnet wird.

Anweisungen

Die Menge aller Java-Anweisungen wird ebenfalls induktiv definiert, wobei die atomare Elemente folgende Anweisungen umfassen.

- beliebige Ausdrucksanweisungen
- Anweisungen `break` und `continue`

[1] Java-Konstrukte, die die Voraussetzungen des Code-Importers erfüllen.

- Anweisung `throw`
- Bedingte Anweisungen mit genau einer Ausdrucksanweisung im `Then`- und `Else`-Block
- `Switch`-Anweisungen mit einer Ausdrucksanweisung in jedem `Case`-Block
- Schleifen mit einer Ausdrucksanweisung im Rumpf
- atomare Try-Catch-Anweisungen

Für die Ausdrucksanweisungen wurde gezeigt, dass deren Semantik durch Transformation in Modelle nicht verändert wird. Die Korrekheit der Übersetzung von anderen Anweisungen muss durch Tests überprüft werden. Im Unterschied zu Ausdrücken, welche alleine durch Verschachtelung konstruiert werden, kann die Menge aller Anweisungen durch den Konstruktor *Verkettung* definiert werden. Als Ergebnis einer Verkettung mehrerer Anweisung entstehen die Blockanweisungen. Es muss also gezeigt werden, dass die Übersetzung einer Blockanweisung bestehend aus beliebigen Anweisungen die Semantik erhalten bleibt. Da Blockanweisung die Reihenfolge der Anweisungen eindeutig definiert und die Übersetzung einzelner Anweisungen als korrekt vorausgesetzt wird, muss sichergestellt werden, dass bei der Übersetzung die Reihenfolge der Teilgraphen stimmt. Dies kann beispielsweise mithilfe der Tests mit mehreren Ausdrucksanweisungen überprüft werden.

Durch strukturelle Analyse konnten einige atomare Konstrukte definiert werden, welche auf Korrekheit der Übersetzung überprüft werden müssen. Im Rahmen der Masterarbeit wurde die Übersetzung einiger dieser Konstrukte auf einem kleinen Beispiel-Projekt getestet. Dazu wurden im Beispiel-Projekt Methoden implementiert, die genau aus einer Anweisung bestehen und genau ein Konstrukt abdecken. Zu jeder Methode wurde ein Modell mit Hilfe des Code-Importers generiert. Abhängig von den Ein- und Ausgabeparametern der Methoden wurden Testszenarien erstellt, welche Eingabeparameter an die Methode und das Modell sowie die Erwartung enthalten. Die Testszenarien wurde auf Methoden in Form von Unit-Tests angewandt. Die generierten Modelle wurden dann mit gleichen Szenarien im jABC-Editor ausgeführt. Danach wurde überprüft, dass die Ergebnisse der Modellausführung den Erwartungen entsprechen.

4 Anwendungsfall

In diesem Kapitel wird eine Applikation zum Qualitätsmanagement im Personennahverkehr QUMA vorgestellt. Obwohl der Stand der Applikation technologisch gesehen fortgeschritten ist, lässt sich QUMA jedoch als Legacy-System bezeichnen. Der Grund dafür ist die Unübersichtlichkeit der Logik-Schicht und die unzureichende sowie veraltete Dokumentation. Im Rahmen dieser Masterarbeit wird ein Geschäftslogik-Modul mittels modellgetriebenen Reengineerings modernisiert. Während des Reengineerings wird der Java-Programmcode mit Hilfe des Code-Importers in ein plattformunabhängiges Format übersetzt. Auf Basis dieses Formates werden jABC-Modelle erstellt, analysiert und manuell angepasst. Aus den Modellen wird danach neuer Quellcode generiert. Während der Integration wird der generierte Code in die Arbeitskopie des Logik-Moduls manuell eingefügt. Den Abschluss des Reengineerings bildet die Verifikation des neuen Systems durch Tests.

4.1 Qualitätsmanagement im Personennahverkehr

Der Zweckverband Verkehrsverbund Rhein-Ruhr (VRR) [59] hat die Aufgabe, die Linien des Schienenpersonennahverkehrs im Gebiet des Verkehrsverbundes zu planen, auszuschreiben und über deren Qualität zu wachen. Der VRR legt in jedem Verkehrsvertrag mit dem Eisenbahnverkehrsunternehmen die Qualität aller Leistungsbestandteile möglichst exakt fest. Kontrolliert werden zum Beispiel die Pünktlichkeit der Züge, der Zustand der Fahrzeuge und der Stationen, die Sitzplatzkapazitäten, die Personale im Zug und weiteres.

Die Eisenbahnunternehmen liefern monatlich Nachweise über ihre Leistungen in Form eines Berichtes. Außerdem überprüfen die Profitester des VRR persönlich das Angebot. Daneben fließen auch die Ergebnisse einer jährlichen Befragung zur Kundenzufriedenheit ein. Der VRR wertet darüber hinaus Beschwerden von Fahrgästen und die Meldungen seiner LinienScouts aus. Werden aus Kundenhinweisen Problembereiche erkannt, kommen dort verstärkt die Profitester zum Einsatz. Am Ende eines Jahres wird Bilanz gezogen. Stellt sich dann heraus, dass die vereinbarten Qualitätsstandards nicht eingehalten wurden, gibt es finanzielle Abzüge.

Zur Kontrolle und Bewertung der erhaltenen Leistungen steht dem VRR eine webbasierte Applikation *QUMA* [44] zur Verfügung. Mithilfe des Systems wird regelmäßig geprüft, ob die Eisenbahnverkehrsunternehmen die vereinbarten Qualitätsstandards auf Ihren Strecken tatsächlich einhalten. Die vereinfachte Arbeitsweise von QUMA wird im Folgenden beschrieben. Die Eisenbahnverkehrsunternehmen spielen Nachweise über gelieferte Leistungen über eine Webschnittstelle in das System ein. Diese monatlich bereitgestellten Leistungsnachweise werden dann von QUMA validiert und statistisch aufbereitet. Anschließend werden sie mit den Daten aus weiteren Erhebungsmethoden (z.B. Profitester-Berichte) zusammengeführt. Auf Basis dieses Datenbestandes bietet QUMA verschiedenste Auswertungsmöglichkeiten und dabei insbesondere die Möglichkeit, die Malus-Berechnung durchzuführen und die Ergebnisse zu dokumentieren. Die verschiedenen Auswertungen werden in Form von Excel-Berichten angeboten.

4.2 Applikation

QUMA ist in der Programmiersprache Java entwickelt worden. Die Architektur der Anwendung wird in Abbildung 4.1 dargestellt. Den Kern der Applikation bilden J2EE Enterprise Java Beans [62], die in einem JBoss Application Server [27] betrieben werden. Für den Zugriff auf die Oracle-Datenbank wurde eine Persistenz-Schicht mit Hilfe von Hibernate [22] implementiert. Die Enterprise Java Beans, in denen die Geschäftslogik implementiert wurde, greifen über die Persistenz-Schicht auf die Daten zu. Der Zugriff auf die Geschäftslogikschicht wird durch eine Web-Präsentationsschicht bereitgestellt. Die Präsentationsschicht besteht aus Komponenten für den Upload von Leistungsnachweisen, Profitester-Stichproben, Komponenten für Datenpflege sowie Komponenten für die Auswertung der Daten und Berichterzeugung. Die Zugriffe geschehen grundsätzlich über HTTP bzw. verschlüsselt über HTTPS.

Die Applikation besteht aus zwei Hauptmodulen: das Modul *quma-logic*, in welchem Datenbearbeitung und Geschäftlogik implementiert sind, und das Modul *quma-web* mit der Implementierung der

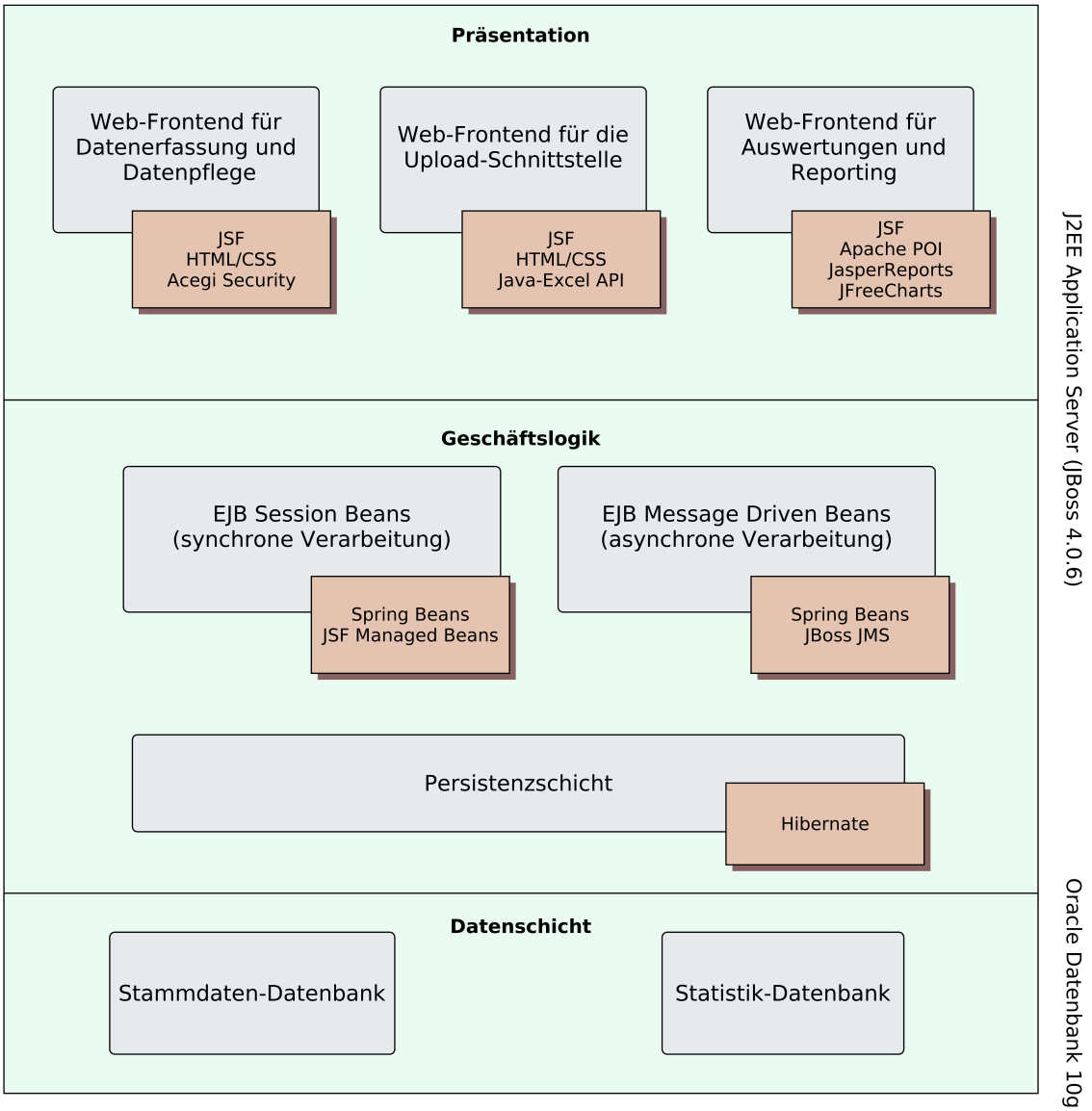


Abbildung 4.1: Architektur von QUMA

Web-Oberfläche. Das Reengineering bezieht sich nur auf das Modul mit der Fachlogik, das Oberflächenmodul bleibt unangetastet. Die Struktur des Moduls *quma-logic* umfasst mehrere Pakete, welche folgende Elemente beinhalten:

- Entity-Objekte
- Data Access Objekte (DAO)
- Services

Entitäten sind Abbildungen der Datenbank-Tabellen. DAOs werden benutzt, um Datenbank-Abfragen zu implementieren und Sammlungen von Entitäten zu liefern. Services repräsentieren die eigentlichen Geschäftsprozesse und greifen über DAOs auf die Entitäten zu. Für das Reengineering wurden einige Services aus dem Fachlogik-Modul ausgewählt. Entitäten und DAOs bleiben in ihrer ursprünglichen Form erhalten.

Die Geschäftslogik in QUMA umfasst verschiedene Prozesse, die in folgende Gruppen aufgeteilt werden können:

- Einspielung von Daten über die Oberfläche in die Datenbank
- Validierung der eingespielten Daten
- Berechnung der Leistungen
- Erstellung der Berichte

Bei der Dateneinspielung handelt es sich um Prozesse, welche die über die Web-Oberfläche angegebene Daten (z.B. Fahrplandaten oder Leistungsnachweise) in das System einlesen. Diese Daten werden nach einer Validierung in die Datenbank eingefügt. Bei der Validierung werden die Daten auf Duplikate, Unstimmigkeiten und Fehler überprüft. Beispielsweise werden doppelte Einträge beseitigt. Falls in einem Leistungsnachweis zwei Datensätze existieren, die sich gegenseitig ausschließen, wird die Einspielung ungültig gemacht und der EVU muss den korrigierten Liefernachweis wiederholt einspielen. Nachdem die Validierung abgeschlossen ist, werden die Leistungen berechnet und in der entsprechenden Datenbank-Tabelle gespeichert. Eine lesbare Darstellung der berechneten Leistungen bieten die Excel-Berichte. Die Berichte enthalten Informationen zu den Konfigurationsparametern der Berechnung und fassen die Daten nach bestimmten Kriterien zusammen (z.B. nach Bundesland oder Verkehrsvertrag).

4.3 Reengineering von QUMA

4.3.1 Auswahl der Geschäftsprozesse

Im Rahmen dieser Masterarbeit wurden Prozesse, die für Berichterzeugung zuständig sind, durch Reengineering modernisiert. Die Prozesse der Dateneinspielung und -Änderungen wurden vom Reengineering ausgeschlossen, weil diese zum größten Teil aus API-Aufrufen des verwendeten Frameworks bestehen und die eigentliche Implementierung in den Frameworks verborgen ist. Da Validierungs- und Berechnungsprozesse durch Vertraulichkeitsvereinbarung geschützt sind, wurde diese nicht verändert.

Bei den Berichterstellungsprozessen werden sogenannte *Exporter* verwendet. Exporter sind Komponenten, die zu einer gegebenen Konfiguration und Daten einen Bericht in Form einer Excel-Datei erzeugen (s. Abbildung 4.2). Eine Konfiguration umfasst einen Zeitraum, die betroffenen Verkehrsunternehmen, eine Auswahl an Qualitätsstandards und Parameter zur Leistungsberechnung. Jeder Bericht besteht aus einer oder mehreren Tabellen (oder Arbeitsblätter), die jeweils einen Qualitätsstandard beschreiben. Zusätzlich zu den Standards, kann ein Bericht eine Übersicht über den Datenbestand bzw. die Konfiguration enthalten. Bei der Erstellung eines Berichtes werden die Daten sortiert und gruppiert. Einzelne Zellen können je nach Standard unterschiedlich formatiert sein. Bei der Erstellung eines Berichts können außerdem einige Berechnungen durchgeführt werden.

Beispielsweise werden Inhalte bestimmter Spalten aufsummiert, Pünktlichkeitsquoten und Anteile der nicht erfüllten Leistungen werden ausgerechnet. Im Weiteren wird erläutert, wie alle Exporter mittels Reengineerings überarbeitet wurden.

4.3.2 Anpassungen der Projektstruktur

Um das Reengineering durchführen zu können, wurden in der Projektstruktur einige Anpassungen vorgenommen. Da beim Reengineering nicht die komplette Applikation betroffen ist, sondern lediglich das Geschäftslogik-Modul, wurde dieses Modul zunächst gesäubert, gebaut und in eine jar-Datei verpackt. Für die Modellbildung wurde ein neues Modul erstellt, in welchem die erzeugten Modelle gespeichert werden. Außerdem wurde eine Arbeitskopie des Moduls erstellt, in der Java-Klassen mit den ausgewählten Prozessen bei der Integration modifiziert werden. Nach diesem Schritt sollte das Original-Modul nicht mehr modifiziert werden.

Die neue Projektstruktur und die Beziehungen zwischen Modulen *original*, *copy* und *models* werden in Abbildung 4.3 dargestellt. Bei der Modellbildung werden Modelle für die ausgewählten Java-Methoden aus dem *original*-Modul generiert. Die erstellten Modelle werden als XML-Dateien im Modul *models* unter `src/main/resources` gespeichert. Die Package-Struktur im Modul *models* entspricht der Package-Struktur des Moduls *original*, sodass Modelle einfacher zu finden sind.

Bei der Code-Generierung wird Quellcode aus Modellen erstellt. Die Java-Dateien werden dabei im *copy*-Modul unter `generated-src/main/java` abgelegt. Somit sind die generierten Klassen von den manuell änderbaren getrennt, denn die kopierten Sources befinden sich im Pfad `src/main/java`. Die Trennung von generierten und nicht-generierten Quelldateien dient nicht nur der Übersichtlichkeit, die generierten Sources können zudem jederzeit problemlos gelöscht und neu generiert werden.

4.3.3 Vorbereitungen des Programmcodes

Für die Modellextraktion wurde das im vorherigen Kapitel vorgestellte jABC-Plugin Code-Importer verwendet. Da die Erstellung der Modelle semiautomatisch abläuft, wurden die Klassen und Methoden, die als Modelle dargestellt werden, im Voraus ausgewählt. Diese Klassen wurden vor dem Anwenden des Plugins angepasst und vorbereitet. Der Programmcode der Exporter musste im Laufe der Masterarbeit an einigen Stellen angepasst werden, um die Generierung von Modellen zu ermöglichen. Durch die Modifikationen, die im Folgenden erläutert werden, hat sich die Semantik des Programmcodes nicht verändert.

- Im Rahmen der Codeanpassung wurde die API der Exporter verfeinert. Beispielsweise wurden alle Exporter-Funktionalitäten zu einem Interface zusammengefasst, sodass alle Im-

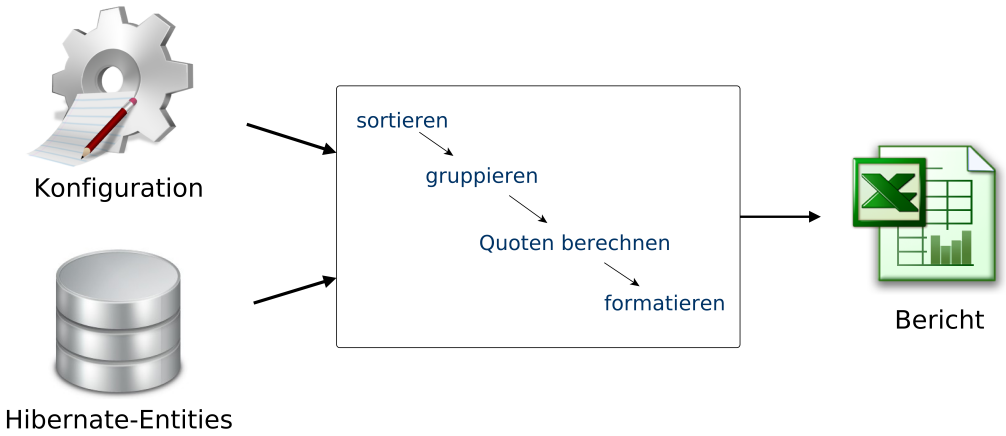


Abbildung 4.2: Funktionsweise der Exporter

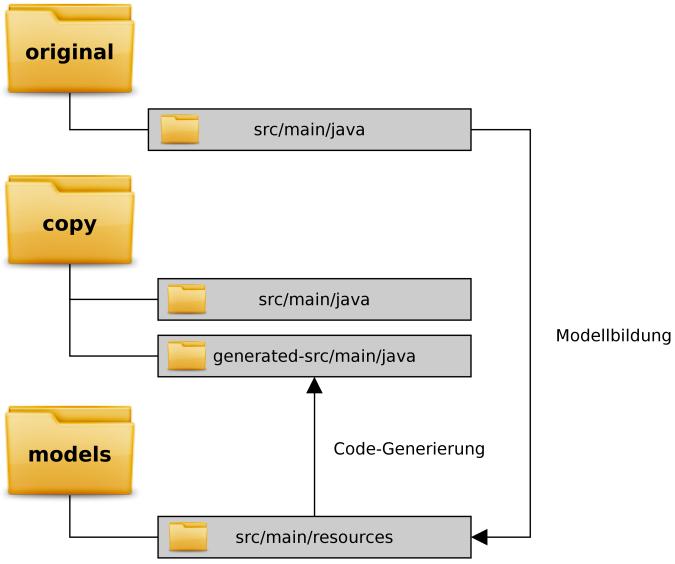


Abbildung 4.3: Anpassungen der Projektstruktur

plementierungen standardisiert sind. Außerdem wurde ein Interface `SheetCreator` erstellt, welches die Verantwortlichkeiten zur Erstellung einer Tabelle zu einem bestimmten Qualitätsstandard übernimmt. Jeder `SheetCreator` besitzt eine Abhängigkeit zu einem `LabelHolder`, dessen Aufgabe die Lokalisierung der verwendeten Labels ist.

- Vor dem Reengineering wurden Methoden-Duplikationen beseitigt. Es wurden mehrere Funktionen identifiziert, die von allen Exportern benutzt werden (z.B. Erstellen einer Tabelle, einer Reihe bzw. Spalte in der Tabelle usw.). Diese wurden vereinheitlicht, zusammengefasst und einer Klasse `ExporterServices` zugeordnet.
- In den Modellen sollen möglichst wenige technische Details auftauchen, wie z.B. Division von `BigDecimal`-Zahlen mit Aufrunden. Mathematische Operationen wurden als Methoden gesammelt und in der Klasse *MathUtils* implementiert. Diese wurden in Modellen als Service-Calls verwendet. Die Trennung zwischen Exporter-Logik und mathematischen Operationen bringt viele Vorteile mit sich. Entscheidet man sich beispielsweise, bei der Division der Zahlen nicht mehr aufzurunden sondern abzurunden, muss lediglich ein SIB ausgetauscht werden.
- Einen Spezialfall der Auslagerung von technischen Details stellt der Zugriff auf die Persistenz-Entitäten dar. Damit Modelle möglichst unabhängig von den Änderungen in der Datenbankstruktur bleiben, werden die Getter-Methoden durch Aufruf eines Zugriff-Services ersetzt. Dieser Service ermöglicht den Zugriff auf eine Instanzvariable eines Objektes, wenn die Variable öffentlich ist oder es eine Getter-Methode zu dieser Variable existiert. Sollten sich die Datenbanktabellen ändern, müssen lediglich die Parameter für den Zugriff-Service angepasst werden.
- Manche Exporter definieren die Spaltenbreite und -höhe als konstante statische Klassenvariablen. Im Rahmen der Code-Anpassung wurden alle Vorkommen von diesen Variablen durch deren Werte ersetzt und die Variablen aus der Klassendefinition entfernt. Dadurch wurde die Anzahl der Eingabeparameter der Graphen deutlich reduziert, denn alle Klassenvariablen werden zu Eingabeparameter der Modellgraphen.
- Alle Java-Konstrukte, die nicht durch den Code-Importer berücksichtigt werden konnten, wurden durch Refactorings angepasst. Zum Beispiel wurden alle innere Klassen in eigene Dateien ausgelagert, anonymen Konstruktoren und Casts wurden Variablennamen gegeben, Übergaben von `Null`-Referenzen wurden durch Overloading ersetzt, die Verwendung der Abkürzung `...` wurde durch Listen umgangen.

- Um unnötig viele automatisch generierten Namen für Kontextvariablen zu vermeiden, wurden komplexe verschachtelte Ausdrücke aufgebrochen und sinnvolle Variablennamen eingeführt.

4.3.4 Ergebnisse der Modellbildung

Insgesamt wurden durch das Reengineering 18 Klassen von 1905 Quellcodezeilen verändert. Die Gesamtanzahl der generierten Modelle beträgt 96. Einige ausgewählte Modelle werden im Folgenden vorgestellt. In allen Abbildungen ist die Ausnahmebehandlung absichtlich entfernt worden, um die Übersichtlichkeit der Modelle zu erhöhen.

NRW-Bericht

Als Beispiel wurde der NRW-Bericht ausgewählt, in dem jährlich eine Übersicht über Leistungen der Verkehrsunternehmen, die in Nordrhein-Westfalen Zuglinien betreiben. Dieser Bericht ist der zentrale Bestandteil aller Berichterstellungsprozesse, denn er umfasst die wichtigsten Qualitätsstandards. Als Teil des Qualitätsberichts SPNV Nordrhein-Westfalen, welcher vom Kompetenzcenter Integraler Taktfahrplan (KC ITF NRW) [29] für ein Kalenderjahr erstellt wird, dient der vom VRR erstellte NRW-Bericht. Der NRW-Bericht betrifft alle Linien, die im Bundesland Nordrhein-Westfalen verlaufen und besteht aus 6 Tabellen:

Konfiguration enthält den Zeitraum des Berichts, Eingrenzung der Linien (z.B. nur Regionalbahn) sowie Konfigurationen zu einzelnen Qualitätsstandards.

Lieferungsübersicht über die verwendeten Datensätze. Unter Anderem werden hier die gültigen Leistungsnachweise einzelner Verkehrsunternehmen aufgelistet.

Ausfall-Ersatz sammelt Ergebnisse zum Qualitätsstandard „Ausfall-Ersatz“, der den Anteil der ausgefallenen Züge beschreibt.

Linien-Pünktlichkeit stellt Resultate der Auswertungen zum Qualitätsstandard Pünktlichkeit einzelner Linien dar.

Messpunkt-Pünktlichkeit umfasst Pünktlichkeiten der Züge an bestimmten Bahnhöfen.

Zugbildung veranschaulicht Ergebnisse zum Qualitätsstandard „Zugbildung“, der die Anzahl der Wagons, Sitzplätze und des Verhältnisses zwischen verschiedenen Sitzplatzklassen überprüft.

Arbeitsblatt „Linien-Pünktlichkeit“

Im Weiteren wird der Prozess Erstellung des Arbeitsblattes zum Qualitätsstandard „Linien-Pünktlichkeit“ vorgestellt. Dieser Prozess wurde in der Klasse `LinienPuenktlichkeitSheetCreator` abgebildet. Dazu wurde die Methode `createSheet` in Modelle mit Hilfe des Code-Importers übersetzt. Im Folgenden wird das Hauptmodell „createSheet“ sowie in diesem Modell referenzierten Untermodelle gezeigt und erläutert. Zunächst wird auf den Arbeitsblatt „Linien-Pünktlichkeit“ eingegangen, der in Abbildung 4.4 abgebildet ist. Die Gruppierung und Sortierung der Daten erfolgt anhand der Spalten „Zeitraum von“, „Zeitraum bis“ und „Linie“. In der Spalte „Fahrten mit mindestens einem Messpunkt“ wird die Anzahl der Fahrten angezeigt, die im gegebenen Zeitraum stattgefunden haben und deren Ankunftszeiten an mindestens einem Bahnhof gemessen wurden. Ein Bahnhof wird als *Messpunkt* bezeichnet, wenn an diesem Bahnhof Ankunftszeiten der Züge gemessen werden. Welche Bahnhöfe als Messpunkte betrachtet werden, wird im Verkehrsvertrag zwischen VRR und dem jeweiligen Verkehrsunternehmen festgelegt. In der Spalte „Anzahl passierter Messpunkte“ werden die Bahnhöfe gezählt, in denen Ankunftszeiten laut Verkehrsvertrag gemessen werden sollen. Als „berichtete Messpunkte“ werden die Messpunkte verstanden, die tatsächlich gemessen wurden.

Zur Berechnung des Standards werden im Wesentlichen zwei Berechnungsverfahren angewandt.

Zeitraum von	Zeitraum bis	Linie	Fahrten mit mind. 1 Messpunkt	Anzahl passierte Messpunkte	Anzahl berichtete Messpunkte	Prozent-PUE: Verspätungs-grenze (in Minuten)	Prozent-PUE: Anzahl verspätete Messpunkte	Prozent-PUE: Pünktlichkeitsquote	Minuten-PUE: Summe durchschn. Verspätungs-minuten	Minuten-PUE: Durchschn. Verspätung einer Fahrt (in Minuten)
Jan 2013	Dez 2013	**RE1**	19703	148594	135628	1	42408	71,36%	36136,31	1,83
Jan 2013	Dez 2013	**RE2**	17591	123137	105398	1	38823	78,47%	29972,94	1,71
Jan 2013	Dez 2013	**RE4**	18764	131794	125002	1	54391	68,73%	47002,78	2,5
Jan 2013	Dez 2013	**RE5**	16762	83810	81958	1	29529	74,76%	27775,19	1,66
Jan 2013	Dez 2013	**RE6**	19548	163559	144929	1	44697	72,61%	51882,94	2,67
Jan 2013	Dez 2013	**RB309**	0	0	0	1	0	-	0	-

Abbildung 4.4: Arbeitsblatt „Linien-Pünktlichkeit“

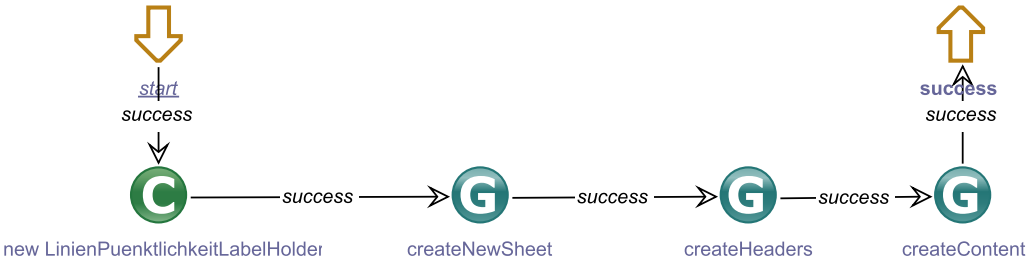


Abbildung 4.5: Modell „Linien-Pünktlichkeit“

Das *Prozentverfahren* zählt den Anteil der Messpunkte, bei denen die Verspätung der Züge die sogenannte *Verspätungsgrenze* überschreitet. Eine Verspätungsgrenze wird in Minuten definiert. Im Arbeitsblatt wird das Prozentverfahren durch Spalten 7 bis 9 beschrieben. Beim *Minutenverfahren* wird die durchschnittliche Verspätung einer Fahrt (in Minuten) über alle Messpunkte ausgerechnet. Das Minutenverfahren lässt sich in Spalten 10 und 11 wiederfinden.

Modell „Linien-Pünktlichkeit“

Das Ergebnis der Übersetzung der Hauptmethode `createSheet` wird in Abbildung 4.5 veranschaulicht. Im Modell „createSheet“ wird im ersten Schritt der `LinienPuenktlichkeitLabelHolder` einen Konstruktor-Aufruf initialisiert. Dieser LabelHolder enthält den Tabellentitel und die Überschriften für das Arbeitsblatt „Linien-Pünktlichkeit“. Im Untermodell „createNewSheet“ wird eine Tabelle erstellt. Der Titel der Tabelle wird aus dem LabelHolder ausgelesen. Im nächsten Schritt wird die Kopfzeile mit Überschriften der Tabelle hinzugefügt. Das letzte Untermodell befasst sich mit Einfügen des Inhalts in die Zeilen nach der Kopfzeile.

Modell „createNewSheet“

Das Erstellen einer neuen Tabelle bzw. eines neuen Tabellenblattes (s. Abbildung 4.6) beinhaltet lediglich zwei Service-Calls. Im ersten Schritt wird der Name der Tabelle aus dem LabelHolder ausgelesen. Dazu wird der Zugriff-Service `PropertyRetriever` verwendet. Im zweiten Schritt wird die neue Tabelle erstellt und der aktuellen Arbeitsmappe hinzugefügt. Dazu wird die API von Apache POI [1] benutzt.

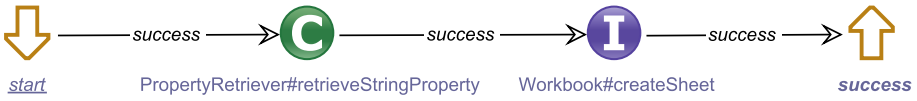


Abbildung 4.6: Modell „createNewSheet“

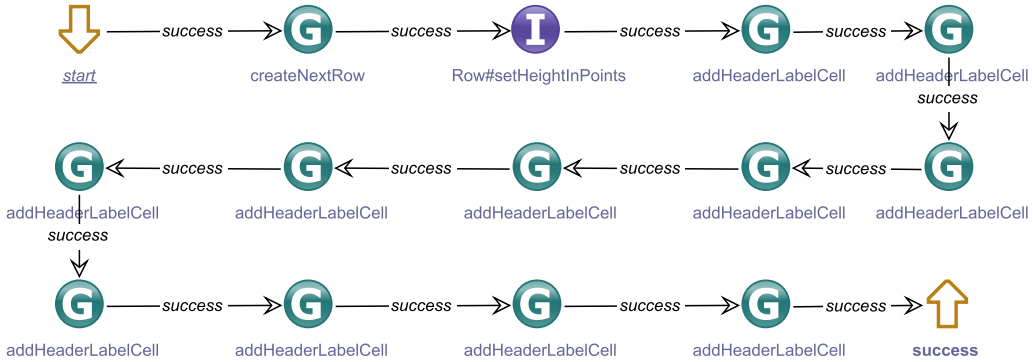


Abbildung 4.7: Modell „createHeaders“

Modell „createHeaders“

Das Modell zum Unterprozess „createHeaders“ (Erstellen der Kopfzeile) wird in Abbildung 4.7 veranschaulicht. Nachdem eine neue Reihe für die Kopfzeile erstellt (Untergraph „createNextRow“) und die Höhe dieser Reihe festgelegt wird, werden die einzelnen Zellen mit Text gefüllt.

Zellen mit Spaltenüberschriften werden mit Hilfe des Untermodells „addHeaderLabelCell“ erstellt. Abbildung 4.8 zeigt den Modellgraphen, der unter Benutzung der API von Apache POI eine Zelle erzeugt, formatiert und mit Inhalt befüllt. Als Erstes wird die Überschrift aus dem `LabelHolder` ausgelesen. Danach wird mit Hilfe des `ColumnHelper` die aktuelle Spalte ermittelt, in welche die neue Überschrift geschrieben wird. Die Reihe, die im Unterprozess „createHeaders“ erzeugt wurde, wird als Eingabe-Parameter dem Start-SIB übergeben. Die Angaben zur Formatierung der Zelle mit der Überschrift stammen ebenfalls aus dem Start-SIB. Das Untermodell „autosizeColumn“ passt die Spaltenbreite der Breite des Textes an.

Modell „createContent“

Abbildung 4.9 zeigt das Modell zum Unterprozess „createContent“, deren Aufgabe das Einfügen des Inhalts in die Tabelle ist. Im ersten Schritt werden die relevanten Datensätze aus der Datenbank gelesen. Dies geschieht über das `LinienPuenktlichkeitsDao`, welches die Liste der Objekte vom Typ `LinienPuenktlichkeit` liefert. In dieser Liste sind nur die Pünktlichkeiten enthalten, die für die gegebene Linie und den gegebenen Zeitraum gelten. Danach wird über die gefundenen Datensätze iteriert und für jede Pünktlichkeit wird eine neue Zeile in der Tabelle erstellt. Jede Zeile beginnt mit den Gruppierungsdaten Zeitraum und Linie. Die Gruppierungsdaten werden unter Verwendung des Service-Graphen „addGruppierungZeitraumLinie“ erstellt und in die Tabelle eingefügt. Für die Spalten „Fahrten mit mindestens einem Messpunkt“, „Anzahl stattgefundener Messpunkte“, „Anzahl berichteter Messpunkte“, „Verspätungsgrenze“ sowie „Anzahl verspäteter Messpunkte“ werden die Inhalte der Entität `LinienPuenktlichkeit` entnommen. Dazu wird der Service-Graph „addBigIntegerCell“ verwendet (s. Abbildung 4.10). Dieser holt den Wert einer

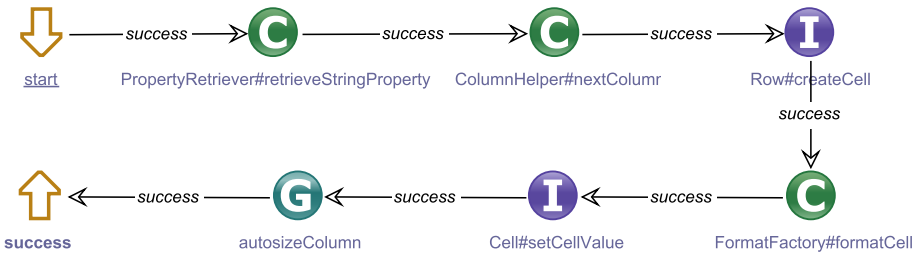


Abbildung 4.8: Modell „addHeaderLabelCell“

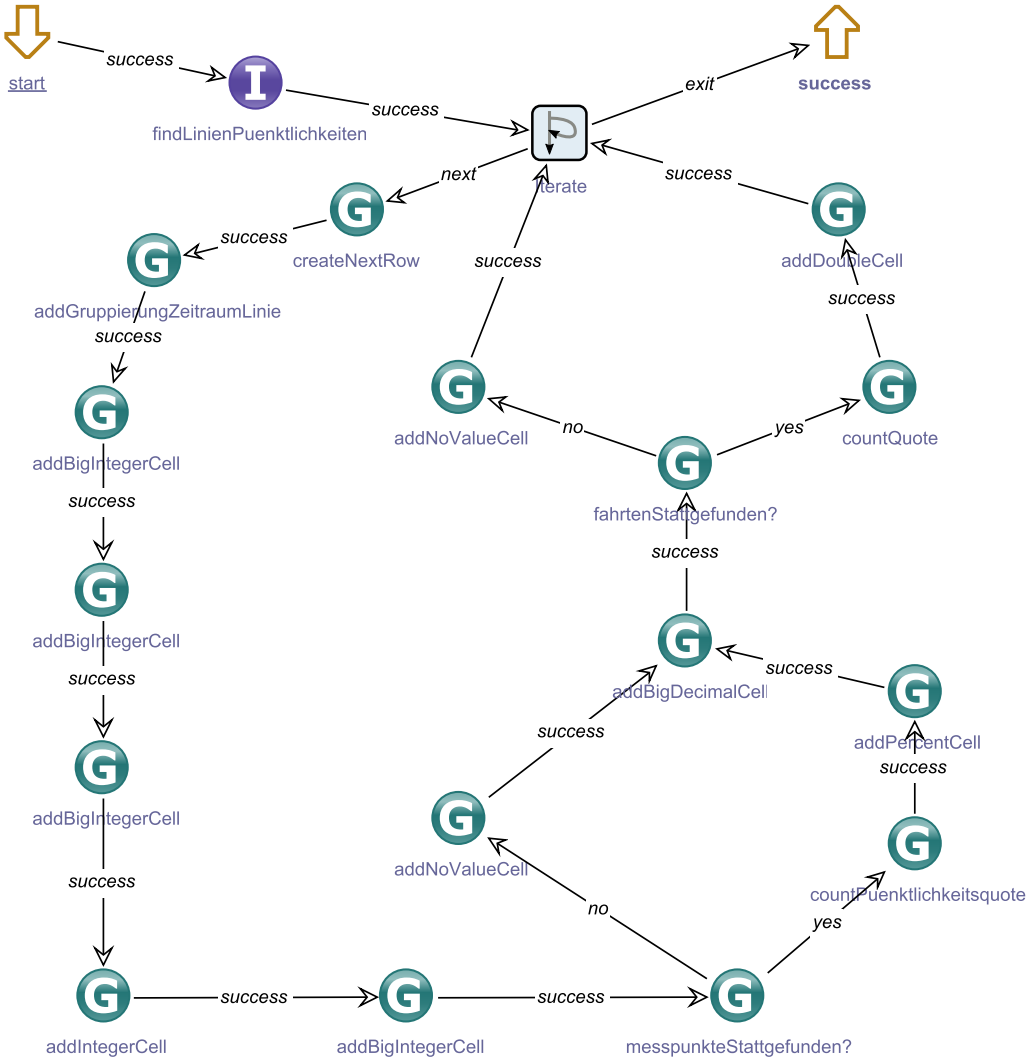


Abbildung 4.9: Modell „createContent“

Instanzvariable aus dem Pünktlichkeitsobjekt und überprüft, ob dieser gleich `null` ist. Wenn der Wert gleich `null` ist, wird eine Zelle mit dem Wert „-“ („ohne Angaben“) erstellt. Sonst wird eine Zahlenzelle erstellt und entsprechend dem angegebenen Format gestaltet.

Nachdem die Inhalte für die ersten sieben Spalten feststehen, wird die sogenannte *Pünktlichkeitsquote* (Prozentanteil der pünktlichen Fahrten) ermittelt (s. Abbildung 4.4). Dazu muss die Anzahl verspäteter Messpunkte durch die Anzahl stattgefundener Messpunkte dividiert werden. Wenn aber keine Messpunkte vorhanden sind, kann die Pünktlichkeitsquote nicht berechnet werden. Deswegen wird vor der Berechnung der Quote überprüft, ob die Anzahl stattgefundener Messpunkte gleich 0 ist. Falls der Service-Graph „messpunkteStattgefunden?“ die Antwort „yes“ ausgibt, wird die Pünktlichkeitsquote berechnet und in eine neue Zelle im Prozent-Format geschrieben. Bei „no“ kann die Quote nicht bestimmt werden, also wird die Zelle dem Wert „-“ befüllt (s. letzte Zeile im Bericht aus Abbildung 4.4). Sonst wird die Quote berechnet und der Tabelle hinzugefügt. Auf ähnliche Weise wird die Berechnung der durchschnittlichen Verspätung einer Fahrt ausgeführt. Die durchschnittliche Verspätung einer Fahrt kann nur dann berechnet werden, wenn die Anzahl der stattgefundenen Fahrten ungleich 0 ist.

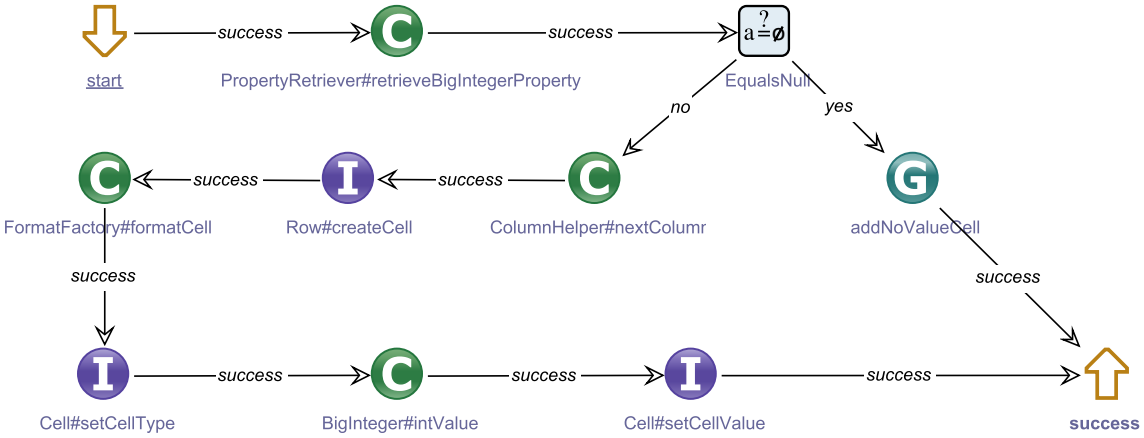


Abbildung 4.10: Modell „addBigIntegerCell“

4.3.5 Analyse der Modelle

Um die Qualität der generierten Modelle zu untersuchen, wurde eine manuelle Modellanalyse durchgeführt. Die Analyse basiert auf folgenden Metriken, die für jeden Modellgraphen manuell ausgerechnet wurden.

Anzahl der Knoten beschreibt die Größe eines Graphen, Input- und Output-SIBs werden mitberechnet. Große Anzahl an Knoten verringert die Übersichtlichkeit und erschwert die Änderbarkeit des Modells.

Anzahl der Kontextvariablen spiegelt die Komplexität und Änderbarkeit eines Modellgraphen wieder. Eine hohe Anzahl an Graphvariablen soll vermieden werden.

Anzahl der Eingabe-Variablen berechnet die Kontextvariablen, die im Input-SIB dem Modellgraphen übergeben werden müssen. Diese Metrik steht im Zusammenhang mit Wiederverwendbarkeit und Benutzerfreundlichkeit der Modellgraphen. Je geringer die Anzahl der Eingabe-Variablen, desto einfacher ist es für Benutzer, den SLG wiederzuverwenden.

Anteil der Service-Calls und Operatoren ist eine Metrik, die den Anteil der elementaren Modellelementen (keine Untergraphen) im Verhältnis zu der Gesamtzahl der Knoten berechnet. Diese Metrik beschreibt den Abstraktionsgrad des Modells und die Abhängigkeit zu den Implementierungsdetails.

Zyklomatische Komplexität bestimmt die Anzahl linear unabhängiger Pfade eines Graphen und wird mit Hilfe der McCabe-Formel [36] $e - n + 2$. Unter e wird hier die Anzahl der Kanten im Graphen und unter n die Anzahl der Knoten verstanden. Je höher die Zahl, desto unübersichtlicher und schwieriger zu testen wird das Modells.

Tiefe der Verschachtelung bezieht sich auf die interne Struktur des Modells. Ein Modellgraph ohne Schleifenkonstrukte erhält die Tiefe der Verschachtelung 1. Jede Schleifenebene erhöht das Ergebnis um Eins.

Vor der Analyse wurden die Modelle in zwei Gruppen unterteilt. Die *High-Level*-Modelle umfassen nur die Service-Graphen, die die Fachlogik der Exporter abbilden. Zu den *Low-Level*-Modellen gehören die Service-Graphen, die von den Fachmodellen verwendet werden, zum Beispiel die Service-Graphen „createNewSheet“, „createNewRow“, „addHeaderCell“. Im Folgenden werden die Ergebnisse der Modellanalyse vorgestellt. Für jede Metrik wurden der minimale und maximale Wert sowie der Durchschnitt über gesamte Gruppe ausgerechnet. Insgesamte wurden 65 High-Level-Modelle und 31 Low-Level-Modelle analysiert. Tabelle 4.1 zeigt die Ergebnisse der Analyse für die High-Level-Modelle und in der Tabelle 4.2 werden die Ergebnisse der gleichen Analysen für die Low-Level-Modelle dargestellt. Resultate der Modellanalyse können als gut bewertet werden.

Durchschnittlich niedrige zyklomatische Komplexität und Tiefe der Verschachtelung über alle Modelltypen weisen auf eine gute Qualität des Quellcodes bzw. Ergebnisse der Code-Anpassungen. Dass der Anteil an Service-Calls und Operatoren in den Low-Level-Modellen relativ hoch ist, war zu erwarten. Denn diese Modelle stellen die Brücke zwischen High-Level-Modellen und der Implementierung dar. Die High-Level-Modelle enthalten deutlich weniger Service-Calls und Operatoren. Einige Maximum-Werte für Knoten- bzw. Variablenanzahl deckten Problemstellen auf, die manuell angepasst wurden.

Metrik	**Minimum**	**Maximum**	**Durchschnitt**
Anzahl der Knoten	4	54	19,04
Anzahl der Kontextvariablen	3	24	9,38
Anzahl der Eingabe-Variablen	1	10	4,57
Anteil der Service-Calls und Operatoren	0,0	0,4	0,19
Zyklomatische Komplexität	1	7	2,3
Tiefe der Verschachtelung	1	3	1,54

Tabelle 4.1: Ergebnisse der Modellanalyse für High-Level Modelle

Metrik	**Minimum**	**Maximum**	**Durchschnitt**
Anzahl der Knoten	4	11	7,17
Anzahl der Kontextvariablen	4	13	6,82
Anzahl der Eingabe-Variablen	2	5	4,52
Anteil der Service-Calls und Operatoren	0,25	0,72	0,63
Zyklomatische Komplexität	1	3	1,47
Tiefe der Verschachtelung	1	1	1

Tabelle 4.2: Ergebnisse der Modellanalyse für Low-Level Modelle

4.3.6 Anpassung der Modelle

Die Ergebnisse der Modellanalyse zeigten einige Problemstellen der generierten Modelle auf. Von den 65 der High-Level-Modelle bestanden 6 Modelle aus mehr als 50 Knoten. Bei dieser hohen Anzahl von Knoten waren die Modelle unübersichtlich. Deshalb wurden aus den entsprechenden Methoden auf der Quellcode-Ebene weitere Methoden extrahiert. Nach der Neugenerierung der Modelle wurden mehr kleinere Modelle erstellt. Eine Verschachtelungstiefe von 3 deutet darauf hin, dass die Modellstruktur eine Schleife innerhalb einer anderen enthält. Bei der Anpassung des Quellcode wurden die betroffenen inneren Schleifen in eine neue Methode verschoben. Dies hat die Übersichtlichkeit der neuen Modelle deutlich verbessert.

Außerdem wurden Methoden angepasst, deren Modelle eine hohe Anzahl der Kontextvariablen (größer 20) erhielten. Die Anzahl der Kontextvariablen wurde reduziert, indem lange Methoden durch Refactoring gesäubert wurden. Dazu wurden die betroffenen Methoden zunächst inhaltlich untersucht. Variablen, die ähnliche Konzepte beschreiben, wurden zusammenfasst und zu einer Gruppe hinzugefügt. Danach wurde versucht, die jeweilige Methode in Teile zu zerlegen, sodass in jedem Teil die Variablen aus möglichst einer Gruppe vorkommen. Diese Teile wurden dann zu einer Untermethode zusammengeführt. Die Variablen wurden somit in jeweiliger Untermethode erzeugt und konnten aus der ursprünglichen Methode entfernt werden.

4.3.7 Code-Generierung

Für die Code-Generierung wurde der Generator jabc4-codegen-java verwendet. Dieser wurde an der technischen Universität Dortmund entwickelt und gehört zu den Standard-Generatoren der

Version 4 von jABC. Wie der Name verrät, wird der Quellcode aus Modellen in Java generiert. Der Generator wurde modellgetrieben mit Hilfe des jABC entwickelt. Die vom Code-Importer erzeugten Modelle werden dem Code-Generator als Eingabe übergeben. Für jedes Modell wird genau eine Java-Klasse erstellt.

Beim Generieren wird zunächst eine Adjazenzmatrix für das gegebene Modell erzeugt. In dieser Matrix werden alle ausgehenden Branches für jedes SIB gespeichert. Außerdem werden der Start-SIB und die Ausgabe-SIBs notiert. Jede generierte Klasse enthält die Methode `execute`, welche die Ausführung des Modells startet. Dabei wird für alle Modelle genau die gleiche Implementiertung generiert. Ausgehend vom Start-SIB wird das Ergebnis der Ausführung im String `_branch_` gespeichert. Für die Kombination aktueller SIB und aktueller Branch wird in der Adjazenzmatrix nach dem Nachfolge-SIB gesucht. Falls der SIB existiert wird dieser ausgeführt und `_branch_` wird aktualisiert. Das Vorgehen wird in Listing 4.1 dargestellt.

Listing 4.1: Die Methode `execute`

```
SIB _currSIB_ = _startSIB_;
    String _branch_ = null;
    while (_currSIB_ != null) {
      _branch_ = _currSIB_.execute();
      if (_currSIB_.isEndSIB()) {
        break;
      }
      final String _succId_ = _adjacencyMap_.get(_currSIB_.getId()).get(
          _branch_);
      if (_succId_ == null) {
        throw new IllegalStateException("SIB '"
            + _currSIB_.getDisplayName()
            + "' returned the branch '" + _branch_ + "'"
            + ", which has no successor.");
      }
      _currSIB_ = _idToContainer_.get(_succId_);
    }
    return _branch_;
```

Die Implementierungen einzelner SIBs des Modells werden als innere Klassen aufgefasst. Je nach Art des SIBs variiert sich die Implementierung von direkten Konstruktor-Aufrufen mit `new` bis Aufrufen eines Untermodells. Zum Beispiel werden für Service-Calls die entsprechende Methode per Java-Reflection aufgerufen. Bei Untergraphen wird eine Instanz der zu diesem Graph generierten Klasse initialisiert und dann die `execute`-Methode aufgerufen. Das Ergebnis der Ausführung aller SIBs kann durch die Methode `getSuccessResult` abgerufen werden. im Anhang B befindet sich die generierte Java-Klasse für das Modell „Linien-Pünktlichkeit“ aus Abbildung 4.5.

4.3.8 Integration

Unter Integration wird im Folgenden das Einfügen des generierten Quellcodes in die Arbeitskopie des Logik-Moduls verstanden. Für eine gegebene Klasse, deren öffentliche Methoden in Modelle übersetzt wurden, kann die Integration durch eine der folgenden Vorgehensweisen durchgeführt werden:

1. Die Klasse bleibt erhalten, deren Getter- und Setter-Methoden sowie alle Konstruktoren bleiben unverändert. Die Implementierungen der Methoden werden durch Modellaufrufe ersetzt.

2. Die Klasse wird gelöscht und alle Aufrufe der Methoden werden durch Modellaufrufe ersetzt.

Obwohl die zweite Strategie für das Reengineering im Allgemeinen besser geeignet ist, kann diese bei QUMA nicht ohne aufwendige Veränderungen eingesetzt werden. Da in den Exportern alle

Instanzvariablen durch Spring-Framework injiziert werden, können die Setter-Methoden nicht entfernt oder in die Modelle integriert werden. Deshalb wurde im Rahmen des Reengineerings von QUMA die erste Strategie verfolgt. Alle Änderungen in den zu übersetzenden Klassen wurden manuell durchgeführt. Im Folgenden werden die Details der Integration am Beispiel erklärt. Listing 4.2 zeigt eine Beispiel-Klasse `AusfallErsatzSheetCreator`, deren öffentliche Methode `createSheet` durch ein Modell dargestellt werden soll.

Listing 4.2: AusfallErsatzSheetCreator vor dem Reengineering

```
public class AusfallErsatzSheetCreator{

  public void createSheet(){
      createHeaders();
      createContent();
      ...
  }

  private void createHeader(){
    ...
  }

  private void createContent(){
    ...
  }

  public void setAusfallErsatzDao(){
    ...
  }

}
```

Nachdem die Methode `createSheet` in ein Modell übersetzt und der daraus generierte Quellcode nach `generated-src` kopiert wurde, kann die Implementierung der Methode durch Ausführung des generierten Codes ersetzt werden. Dabei können alle in der Methode verwendeten privaten Methoden aus der Klasse entfernt werden, denn diese werden durch Untermodelle repräsentiert. Die Instanz des Modells wird als Instanzvariable deklariert und durch das Injection-Framework initialisiert. Das Ergebnis des Reengineerings für die Klasse `AusfallErsatzSheetCreator` wird im Listing 4.3 dargestellt.

Listing 4.3: AusfallErsatzSheetCreator nach dem Reengineering

```
public class AusfallErsatzSheetCreator{

  @Inject
  private AusfallErsatzSheetCreator_createSheet createSheetModel;

  public void createSheet(){
      createSheetModel.execute();
  }

  public void setAusfallErsatzDao(){
    ...
  }

}
```

Falls die zu übersetzende Methode einen Rückgabewert besitzt, wird bei der Integration die entsprechende `Return`-Anweisung hinzugefügt. Wenn das Modelle eine Ausnahme weiterleiten kann, muss die Ausnahmebehandlung implementiert werden.

4.3.9 Verifikation

Bei der Verifikation wird geprüft, ob die Übersetzung des Quellcodes in Modelle und die nachfolgende Code-Generierung das Verhalten des Systems nicht verändert haben. Um den Vergleich zwischen dem alten und neuen Verhalten wurden Testfälle für das existierende System erstellt. Alle Testfälle wurden als Black-Box-Tests entwickelt, diese definieren mehrere Kombinationen von Eingaben und Ausgabe-Erwartungen. Die verwendeten Szenarien deckten vor allem die Randfälle ab. Es sollten beispielsweise Berichte erzeugt werden, auch wenn keine Daten vorhanden sind. Der Zugriff auf die Datenbank wurde durch Mock-Objekte simuliert. Außer Angaben zu den Daten gehören Informationen zu Formatierungen einzelner Zellen ebenfalls zu der Erwartung. Als Erwartung diente eine Tabellenbeschreibung mit Hilfe der Apache POI API. Die Tests wurden zunächst auf dem alten und dann auf dem neuen System ausgeführt. Da die Ergebnisse der Tests auf beiden Systemen gleich waren, kann angenommen werden, dass sich das Verhalten des Systems durch das Reengineering nicht verändert hat.

4.4 Diskussion der Ergebnisse des Reengineerings

In diesem Abschnitt werden die Ergebnisse des Reengineerings evaluiert. Folgende Aspekte, werden betrachtet: Übersichtlichkeit, Verständlichkeit und Dokumentation, Wartbarkeit und Änderbarkeit, Portabilität und Performance. Alle Aspekte werden auf dem System nach dem Reengineering untersucht. Anhand von Beispielen wird gezeigt, welche Verbesserungen durch Reengineering erreicht wurden. Außerdem werden einige möglichen Problemstellen identifiziert.

Übersichtlichkeit

Dadurch, dass nur Teil der Applikation durch Reengineering verändert wurde, hat sich die Struktur des Gesamtsystems nicht verändert. Die generierten Java-Klassen werden im getrennten Source-Ordner abgelegt, sodass diese bei Bedarf gelöscht und neu generiert werden können. Durch eine wohldefinierte API wird festgelegt, an welchen Stellen Modelle aufgerufen werden: Modelle werden in der Methode `createSheet` einer Klasse `SheetCreator` initilisiert und gestartet.

Mit Hilfe der Modelle ist die Übersichtlichkeit der Prozesse deutlich gestiegen. Vor allem für Methoden mit mehr als vier Parametern ist die Darstellung als Graphen viel übersichtlicher (s. Abbildung 4.11). Um einen Überblick über Prozesse zu verschaffen, muss man im Code die gesamten Parameter mitlesen. Dabei reicht es zu wissen, welche Schritte durchgeführt werden, ohne konkrete Argumente. Dazu Abfolge von SIBs mit entsprechenden Namen einen bessere Übersicht über die Prozessschritte als vorherige Quellcode.

Ein anderes Beispiel zeigt, dass die Auslegung der Modellknoten zur Übersichtlichkeit viel beitragen kann. Der Bericht aus Abbildung 4.12 enthält zwei Kopfzeilen, wobei die Anzahl der Zellen in beiden Kopfzeile unterschiedlich ist. Im Quellcode wird für jede Zelle ein Methodenaufruf verwendet und der Anfang einer neuen Kopfzeile durch Kommentar erkennbar gemacht wird. Im

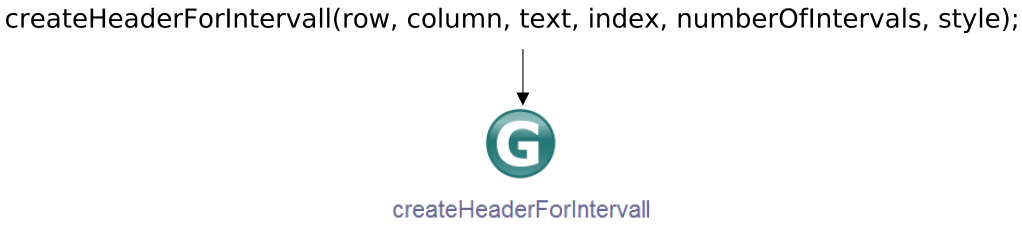


Abbildung 4.11: Übersichtlichkeit eines SIBs gegenüber einem Methodenaufruf

		Soll		Ist: gefahrene Fahrten			Fahrten mit Abweichungen in der			Ausgefallene Sitzplätze (Basis: Sitzplatz-KM)			
Linie	Monat	bestellte ZugKM		ZugKM	Sitzplatz-KM		ZugKM	davon mit Sitzplatzabweichung (ZugKM)	davon mit Komfortabweichung (ZugKM)	Anzahl Sitzplatz-KM	Anteil am IST	Gewichtung	entspricht in ZugKM (relevant für Summe Abzüge)
Test001	Feb 2008	1.000,00		1.000,00	10.000,00		200,00	100,00	200,00	2.000,00	20,00%	200,00%	400,00

Abbildung 4.12: Bericht mit zwei Kopfzeilen

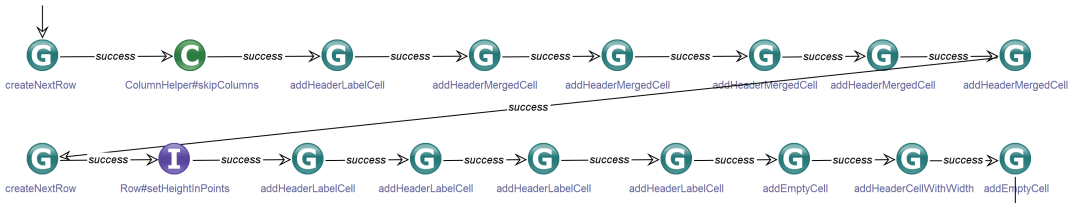


Abbildung 4.13: Modell für zwei Kopfzeilen

Modell kann man die Knoten so anreihen, dass alle Zellen einer Kopfzeile auf der gleichen Gerade angeordnet sind. Dies wird in Abbildung 4.13 veranschaulicht.

Dokumentation

Im Unterschied zu Applikation vor dem Reengineering ist die Dokumentation im neuen System direkt in Modellen „integriert“. Man kann die Modellelemente genau so wie den Quellcode mit Kommentaren und Annotationen anreichern. Außerdem helfen die Icons die Funktionalität eines Knotens zu verbildlichen. Abbildung 4.14 zeigt einige in QUMA verwendeten Untermodelle mit neuen Icons, die die Bedeutung einzelner Unterprozesse widerspiegeln. Ohne fundierte Fachkenntnisse kann der Abbildung entnommen werden, dass in diesem Modell ein neues Tabellenblatt erstellt wird. Diese Blatt besteht aus genau einer Zeile. Diese Zeile enthält eine Zelle mit Text, eine Zelle mit Ganzzahl, eine Zelle mit einer Dezimalzahl und eine Zelle mit einer Prozentzahl.

In der Implementierung ist noch nicht vorgesehen, dass Modelle beim Erstellen mit einem Icon versehen werden. Icons müssen nach der Modellbildung manuell eingefügt werden. Allerdings kann der Generierungsprozess erweitert werden, sodass jedem Modell ein Icon zuordnet wird.

Änderbarkeit und Wartung

Wenn Prozesse als Modellgraphen implementiert werden, sind sie einfach zu ändern. Das Entfernen bzw. Verschieben einer Spalte im Bericht kann mit wenigen Klicks erledigt werden (s. Abbildung 4.15). Das Hinzufügen einer neuen Spalte oder Zeile stellt ebenfalls kein Problem dar, denn es können schon existierende Modelle benutzt werden. Dazu muss man lediglich das passende Modell finden, diese per Drag-and-Drop auf die Oberfläche platzieren und mit anderen Knoten durch Kanten verbinden.

Problematisch können die Änderungen an der API sein. Vor allem wenn die Klassen, in denen

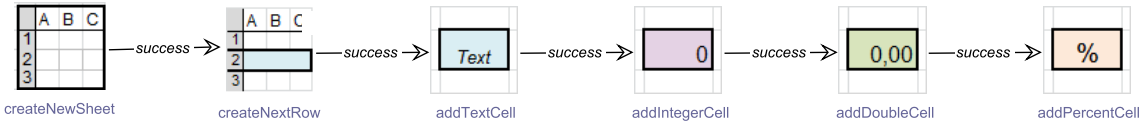


Abbildung 4.14: Mögliche Icons für einige Untermodelle in QUMA

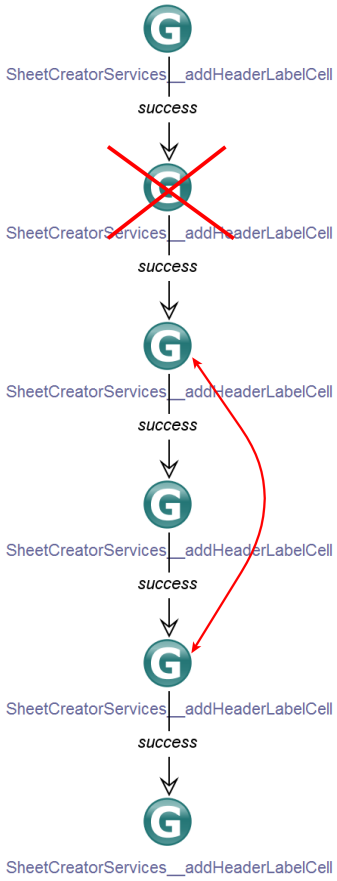


Abbildung 4.15: Änderbarkeit von Modellen

Modelle aufgerufen werden, verändert werden, ist Vorsicht geboten. Falls Änderungen die Web-Oberfläche betreffen, müssen die Modelle nicht angepasst werden. Die Exporter werden von Services aus dem Web-Modul über die API aufgerufen, die durch Reengineering nicht modifiziert wurde. Wenn die Datenbank-Schicht umformt wird, könnten die Modelle betroffen werden. Allerdings werden die Änderungen keinen großen Charakter haben, da der Zugriff auf die Daten-Entitäten nicht direkt sondern über einen Zugriff-Service durchgeführt werden.

Erweiterbarkeit

Falls ein neuer Bericht (zum Beispiel für ein neues Bundesland) oder ein Bericht um einen neue Tabelle erweitert werden muss, müssen neue Modelle erstellt werden. Da die Funktionalitäten, wie Erstellen einer leeren Tabelle, einer neuen Reihe usw. schon als Modelle existieren, können die wiederverwendet werden. Dazu müssen lediglich die Ein- und Ausgabeparameter angepasst werden. Wenn ein neuer Bericht komplexere Vorgänge abbilden soll, kann ein Fachexperte bei der Modellierung um Unterstützung gebeten werden. Abläufe, die als Graphen dargestellt werden, können von den Fachexperten besser verstanden werden, als der Quellcode. Zeit kann gespart und Probleme können frühzeitig gelöst werden.

Portabilität

Da die erstellten Modelle den Quellcode sehr nah abbilden, besitzen diese die gleichen Abhängigkeiten zu externen Frameworks wie der Programmcode. Das bedeutet, dass die Portabilität genau so gewährleistet wird, wie diese im Quellcode gewährleistet wurde. Beispielsweise wäre eine Portierung von Apache POI auf ein anderes Framework problematisch, da fast alle Modelle Klassen

aus der API von Apache POI benutzen. Wenn die Portierung das Hauptziel des modellgetriebenen Reengineerings wäre, müsste man den Quellcode vorher noch mehr aufbereiten. Man müsste dann die Klassen aus der API zunächst in eigene Klassen kapseln und erst dann die Modelle generieren lassen. Danach könnte man die Wrapper modifizieren und ein neues Framework verwenden, ohne dass die Modelle geändert werden müssen.

Performance

Die Performance und Effizienz des Systems nach dem Reengineering hängt im wesentlichen vom verwendeten Code-Generator ab. Wenn dieser eine effiziente Implementierung anbietet, sollte das neue System nicht langsamer sein als das alte. Um eine verbindliche Aussage über den Einfluss von Modellen auf die Performance zu machen, müssen spezielle Test durchgeführt werden.

5 Verwandte Arbeiten

Es existiert eine Vielzahl an Forschungsarbeiten und Werkzeugen, die zum modellgetriebenen Reengineering eingesetzt werden können bzw. bei einigen Aufgaben unterstützen. Die Projekte unterscheiden sich in den verwendeten Modellen und in der Art, wie Modelle erstellt werden. Im Folgenden werden vier Ansätze (bzw. Tools) vorgestellt, die als repräsentative Vertreter unterschiedlicher Gruppen dienen. Der erste Ansatz stellt ein Bottom-Up-Verfahren zur Migration einer kompletten Anwendung dar. Im zweiten Abschnitt werden auf Reengineering-Projekte mit TGraphen eingegangen. Das Projekt *MoDisco* wird verwendet, um Ecore-Modelle aus Java-Quellcode zu erstellen und verwalten. Mit Hilfe der Bibliothek *Gra2MoL* können Programmiersprachkonstrukte auf beliebig allgemeine Modellelemente abgebildet werden.

5.1 Modellgetriebene Software-Migration

Am ähnlichsten ist in dieser Arbeit vorgestellte Ansatz dem Bottom-Up-Ansatz aus [61]. In dieser Arbeit wurde eine umfangreiche C++-Applikation (etwa 97.000 LoC) nach Java migriert. Während der Migration wurde der Programmcode in Modelle transformiert. Hierfür wurde zunächst ein C/C++-Parser angewandt, der die textuelle Repräsentation des Quellcodes in eine maschinenverarbeitbare Form überführt. Als grundlegende Datenstruktur für die Modellbildung dienten die annotierten abstrakten Syntaxbäume. Diese wurden über das konzipierte Backend in eine plattformunabhängige Beschreibung, also eine Modellbeschreibung, übersetzt. Das Ergebnis wurde mit einer Vielzahl an Informationen angereichert, beispielsweise wurde die Zeile in der Quelldatei notiert.

Mit Hilfe des Modellierungswerkzeugs jABC wurden die generierten Beschreibungen eingelesen und als Kontrollflussgraphen dargestellt. Jeder Knoten im Modellgraphen repräsentierte eine Anweisung aus dem Quellcode. Ein typisches Modell nach der Übersetzung wird in Abbildung 5.1 veranschaulicht. Der Abbildung kann entnommen werden, dass die generierten Modelle sehr feingranulare Darstellung des Quellcodes anbieten.

Durch Verwendung des jABC-Editors und einer eingebauten Datenbank mit allen generierten Modellen wurden die erzeugten Modelle semiautomatisch verfeinert bzw. zusammenfasst. Dabei stand man allerdings einer sehr großen Menge an Informationen gegenüber, die durch gezielte Analysen und Abstraktionen verkleinert werden musste[61]. Um auf der Informationsmenge eine sinnvolle Abstraktion zu definieren und damit diese stark einzuschränken, hat der Autor der Arbeit eine API-basierte Abstraktion entwickelt, umgesetzt und getestet.

5.2 Reengineering mit TGraphen

Bei TGraphen, zuerst vorgestellt in [48], handelt es sich um typisierte, attributierte, angeordnete und gerichtete Graphen. Den Knoten und Kanten eines TGraphen können Typen aus dem vordefinierten Schema zugeordnet werden. Alle Knoten und Kanten können Attribut-Wert-Paare beinhalten. Innerhalb eines Graphen existieren Anordnungen der Graphelemente untereinander. Eine weitere Anordnung existiert fur Kanten und die TGraphen selbst können angeordnet werden.

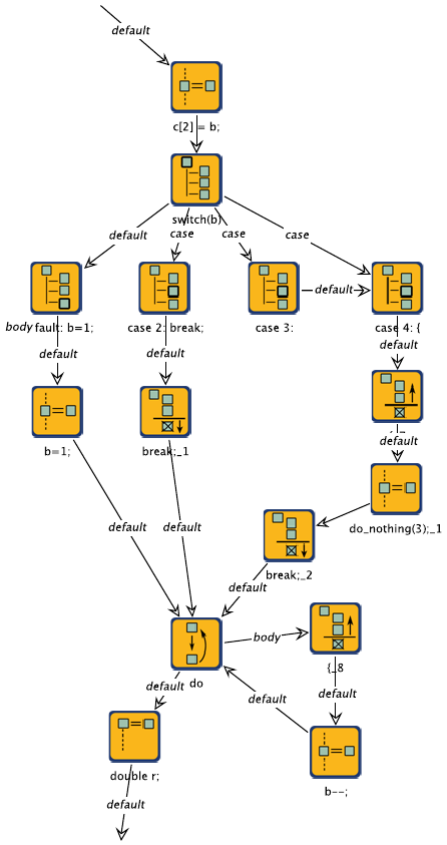


Abbildung 5.1: Ein Modell des Bottom-Up-Ansatzes [61, S. 161]

Abbildung 5.2 zeigt ein Beispiel für einen TGraphen, der Beziehungen in einem Strassensystem darstellt. In der modellgetriebenen Entwicklung und beim modellgetriebenen Reengineering werden die TGraphen als Modelle eingesetzt [12, 20]. Für die Manipulation, Traversierung und Transformation von TGraphen dient das Framework *GraLab* [12]. JGraLab (Java-Version des GraLab) bietet eine visuelle Darstellung der TGraphen und enthält einen Code-Generator. Außerdem kann man mit Hilfe der Erweiterung *Java-Extractor* [25] den Java-Quellcode in TGraphen überführen. Java-Extractor kann Java-Quellcode in einen UML-ähnlichen TGraph automatisch konvertieren. Eine Abbildung des Kontrollflusses ist noch nicht möglich.

Im Unterschied zu abstrakten Syntaxbäumen können die TGraphen zusätzliche Informationen über Software-Systeme aufnehmen, wie z.B. Beziehungen zwischen Objekten und Klassen. Durch Abfragen an generierte TGraphen können programmanalytische Aufgaben erledigt werden. Für die Abfrage von Information aus TGraphen steht die Abfragesprache *GReQL* [12] zur Verfügung, die über eine eigene Syntax verfügt. Mit Hilfe dieser Abfragesprache können Software-Metriken berechnet werden, um den Quellcode zu analysieren. Da TGraphen Referenzen auf Quellcode besitzen, können durch Abfragen Code-Anpassungen durchgeführt werden. Die Clustering-Funktion erlaubt es zu analysieren, welche Softwarekomponenten zum gleichen Modul gehören.

Die Liste der Projekte, in denen TGraphen für das Reengineering verwendet wurden, ist lang. Zu den aktuellsten zählt das Projekt SOAMIG [16], welches die Transformation von Legacy-Systemen in Service-orientierte Architekturen ermöglicht. Im Projekt wurde Cobol- nach Java-Programmcode transformiert und zugleich wurden aus dem Java-Programmcode autonome Services erstellt.

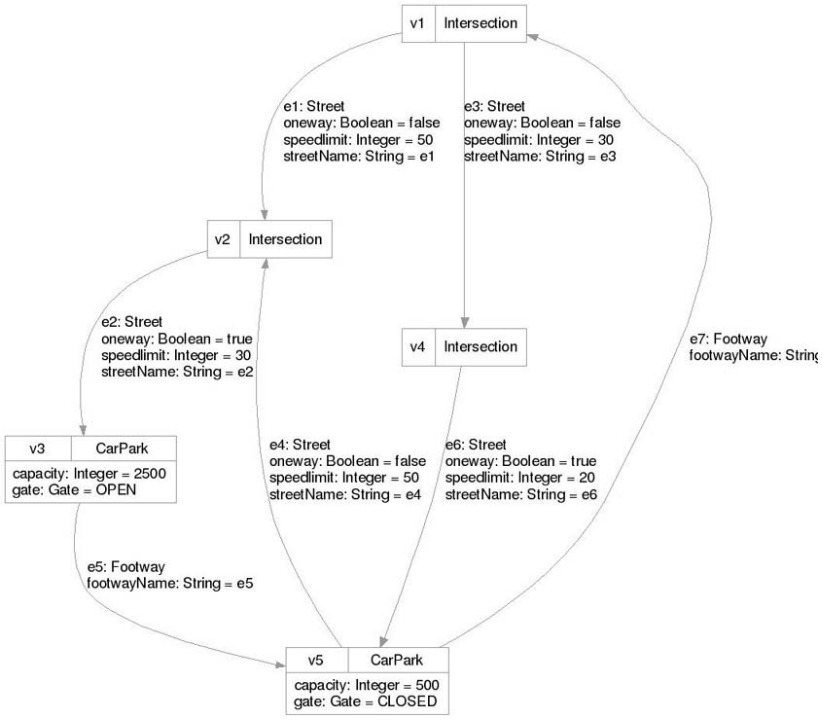


Abbildung 5.2: Beispiel für einen TGraphen [12]

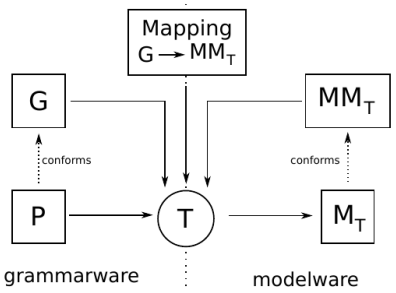


Abbildung 5.3: Funktionsweise des Gra2MoL

5.3 Gra2MoL

Gra2Mol [24] ist eine domänenspezifische Sprache, die eingesetzt wird, um Modelle aus dem Quellcode einer beliebigen General-Purpose-Programmiersprache zu extrahieren. Diese DSL implementiert eine Brücke zwischen der Grammatik einer Programmiersprache (auch *grammarware* genannt) und den Elementen eines Modells (*modelware*). Der Hauptbestandteil von GRa2MoL sind die sogenannten *Mappings* von den Elementen der Quellcode-Grammatik auf die Elemente einer Modellierungssprache. Ausgehen von den Mapping-Regeln werden Modelle aus dem Source-Code automatisch generiert. Der erste Schritt in der Code-zu-Modell-Tranformation ist der Aufbau der abstrakten Syntaxbäumen. Danach werden diese ASTs durchlaufen, um für die Erstellung der Modelle benötigte Informationen zu sammeln. Welche Informationen aus dem AST zur Erstellung eines Modellelements verwendet werden sollen, kann man in Form von Queries in den Mappings definieren.

Abbildung 5.3 zeigt den Ablauf einer Text-zu-Modell-Tranformation mit Hilfe von Gra2MoL. Als Eingabe wird das Programmcode P und deren Grammatik G verwendet. Die Ausgabe der Tranfor-

mation T ist das Modell M_T mit dem Metamodell MM_T erstellt. Bei der Tranformation kommen Regeln aus dem Mapping $G \rightarrow MM_T$ zum Einsatz. Jede Regel besteht aus vier Teilen:

from spezifiziert einen nicht-terminalen Knoten im AST

to beschreibt das Ziel-Element im Modell

queries beinhalten die Queries auf den AST, die ausgehend vom aktuellen Knoten ausgeführt werden

mapping umfasst eine Menge von Abbildungen von AST-Eigenschaften auf Eigenschaften des Modell-Elements

Ein Beispiel für eine Regel wird in Abbildung 5.4 gezeigt. In diesem Beispiel wird eine Methodendeklaration zum Modellelement `Method` transformiert. Die Parameter einer Methode werden auf den Knoten `Parameter` abgebildet.

Die erste Version des Gra2MoL wurde in 2008 veröffentlicht. Seitdem wurden einige zusätzliche Features hinzugefügt, wie z.B. die Möglichkeit, eine Regel in einer anderen zu benutzen oder eine Regel zu überspringen. Außerdem können die Queries oder Mappings um benutzerspezifischen Operatoren erweitert werden. Gra2MoL wurde in verschiedenen Projekten mit unterschiedlichen Programmiersprachen eingesetzt. Beispielsweise wurden in [8] die Oracle Forms von PL/SQL auf Java migriert.

5.4 MoDisco

MoDisco [6] ist ein Eclipse-basiertes Tool für modellgetriebenes Reengineering, welches seit 2006 von AtlanMod-Forschungsteam in Kooperation mit Mia-Software entwickelt wird. Mithilfe des Werkzeugs kann die Modernisierung eines Systems in drei Phasen durchgeführt werden, welche in Abbildung 5.5 visualisiert werden. In der Phase **Extraktion von Informationen** kann einer der 17 im Framework enthaltenen *Discoverer* angewandt werden. Die Discoverer werden primär bei Java Web Applications eingesetzt und können Java-Quellcode, J2EE (z.B. jsp-Webpages), XML und UML Dateien parsen. Durch Erkennung von logischen und physischen Entitäten können verschiedene Modelle für die gleiche Datei erzeugt werden. Für den Java-Quellcode werden Modelle basierend auf dem Ecore-Metamodell [14] generiert. Ein Beispiel-Modell wird in Abbildung 5.6 dargestellt.

In der zweiten Phase **Verstehen** werden die vielfältigen Modelle mithilfe der *Model Browser* untersucht. Zusätzlich zum Visualisieren, Anordnen und Filtern können Anfragen an Modelle gestellt werden. Durch Anfragen kann beispielsweise ermittelt werden, welche Java-Quelldateien bestimmte Bibliotheken verwenden, um diese Abhängigkeiten zu entfernen oder ersetzen. Alle Ecore-Modelle werden mit dem Quellcode verlinkt. Mithilfe der *Modell-Facetten* können angepasste Sichten auf

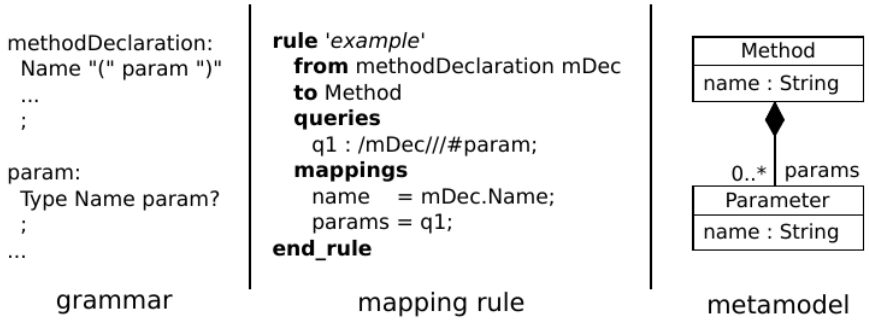


Abbildung 5.4: Eine einfache Gra2MoL-Regel [8]

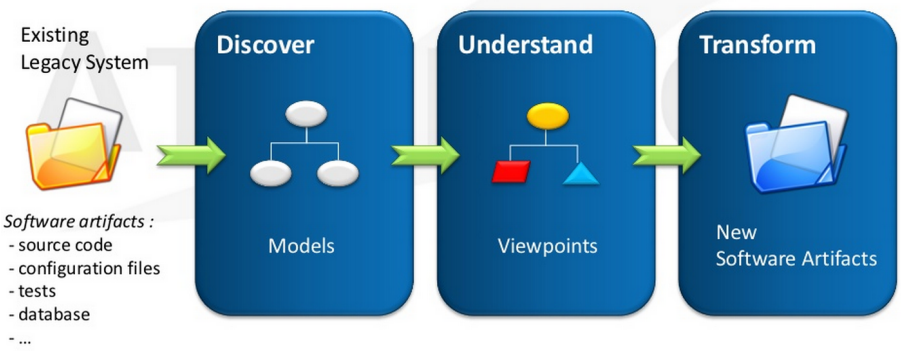


Abbildung 5.5: Drei Phasen des Reengineerings in MoDisco [6]

ein Modell erstellt werden. Neue Elemente, Attribute oder Referenzen könne hinzugefügt werden. In der letzten Phase, der **Transformation**, werden gefilterte und angepasste Modelle in neue Artefakte transformiert, um die Modernisierung zu durchzuführen. Unter anderem kann neuer Quellcode aus Ecore-Modellen generiert werden.

Zu den Stärken des Tools gehören hohe Anpassbarkeit und Erweiterungsmöglichkeiten durch eigene Eclipse-Plugins. Verschiedene Discoverer verschaffen die abstrakte Sicht auf unterschiedliche Aspekte einer Applikation und bringen bei Aufgaben wie Architektur-Rekonstruktion viele Vorteile mit sich. Durch die Ausrichtung „Werkzeugkasten statt Werkzeug“ kann die Vielfältigkeit der Werkzeuge auf manche Benutzer allerdings überfordernd wirken. Zu den Anwendungsmöglichkeiten für MoDisco zählen Einführung neuer Standards, Migration auf neue Versionen (z.B. JUnit3 zu JUnit4, EJB2 zu EJB3), Auffinden der Anti-Muster bzw. Problemstellen, Rekonstruktion der Architektur.

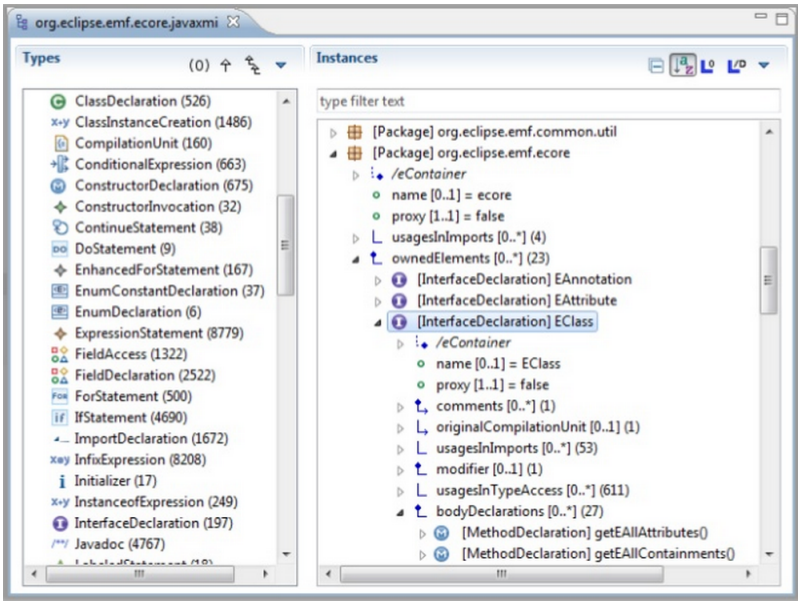


Abbildung 5.6: Ecore-Modelle in MoDisco [6]

6 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und erläutert. Abschließend werden Ideen für Erweiterungen und Verbesserungen des vorgestellten Ansatzes in einem Ausblick kurz vorgestellt.

6.1 Zusammenfassung

Durch ständige Anpassungen und Erweiterungen entstehen aus existierenden Anwendungen sogenannte Legacy-Systeme mit veralteten Technologien und unübersichtlicher Struktur. Um Legacy-Software den modernen Anforderungen anzunähern, können Altsysteme durch Reengineering umstrukturiert werden. Beim Reengineering wird der Programmcode in abstrakten formalen Modellen abgebildet, die einen Überblick über verlorene Zusammenhänge wiederherstellen können. Nach der Modellanalyse können diese verfeinert werden. Anschließend wird neuer Quellcode generiert oder manuell implementiert. Beim modellgetriebenen Reengineering werden Modelle zu den Hauptartefakten des neuen Systems. Alle folgende Änderungen und Erweiterungen werden auf der Modellebene ausgeführt.

Im Fokus dieser Masterarbeit stand die Erarbeitung eines Verfahrens zum Generieren von Modellen aus dem existierenden Quellcode. Bei der Umsetzung des Verfahrens wird ein semiautomatischer Ansatz verfolgt. Dem Benutzer wird die Auswahl der Programmteile mit den zu modernisierenden Geschäftsprozessen überlassen. Die Granularität der generierten Modelle ist ebenfalls manuell einstellbar. Die Transformation der Quellcode-Abschnitte zu Service Logic Graphen verläuft automatisch. Dabei wird jedes Konstrukt der Programmiersprache auf ein Modellelement abgebildet. Die Verbindungen zwischen den Modellknoten entsprechen dem Kontrollfluss des Programms.

Die prototypische Implementierung des Ansatzes wurde in einer existierenden Web-Applikation eingesetzt. Mit Hilfe des Prototyps wurden einige Geschäftsprozesse aus der Anwendung in Modelle transformiert und durch Reengineering modernisiert. Die erstellten Modelle sind ausführbar und von guter Qualität, aus diesen konnte direkt neuer Code generiert werden. Nach der Code-Generierung wurde durch Tests bestätigt, dass sich das Verhalten des Systems durch das Reengineering nicht verändert hat. Während der Evaluation konnte festgestellt werden, dass die Übersichtlichkeit und Wartbarkeit der Prozesse durch die Modellbildung deutlich verbessert wurden. Die modernisierte Applikation kann einfacher erweitert werden, auch von Fachexperten ohne tiefergehenden Programmierkenntnissen. Außerdem stellte sich heraus, dass die Qualität der Modelle von den Eigenschaften des verwendeten Quellcodes abhängig ist. Zwar ist ein manueller Aufwand für die Aufbereitung des Programmcodes notwendig, doch dadurch wird Qualität des neuen Software-Systems deutlich erhöht.

6.2 Ausblick

Abschließend werden einige Erweiterungen des Reengineering-Prozesses vorgestellt, die helfen können, das Verfahren zu verbessern.

Erweiterungen des Code-Importers

Der Code-Importer ist eine prototypische Umsetzung des entwickelten Verfahrens und hat ein großes Erweiterungspotenzial. Der Importer kann nur für Java-Applikationen angewandt werden. Da die Implementierung auf abstrakten Syntaxbäumen aufbaut bzw. verwendet, kann der Code-Importer für weitere Programmiersprachen verfügbar gemacht werden. Dazu müssen lediglich die Syntaxbäume für die jeweilige Sprache aufgebaut werden, beispielsweise durch Verwendung von geeigneten Werkzeugen oder Bibliotheken.

Im Abschnitt 3.4 wurde bereits aufgezeigt, dass bei der Modellbildung nicht alle Java-Konstrukte berücksichtigt werden. Einige Konstrukte, wie zum Beispiel Annotationen, können wichtige Informationen über die Funktionsweise einer Methode enthalten. Diese sollten in die Implementierung des Code-Imports miteinbezogen werden. Die Generierung der Variablennamen für verschachtelte Ausdrücke kann verbessert werden. Ein Variablenname kann aus dem Typ der Variable abgeleitet werden, bei den Getter- bzw. Setter-Methoden ergibt sich eine neue Variable aus dem Methodennamen.

Während der Evaluation von erstellten Modellen (s. Abschnitt 4.4) wurde festgestellt, dass diese einige plattform-abhängige Elemente enthalten. Die Qualität der Modelle kann verbessert werden, wenn die plattform-spezifischen Details vom Code-Importer durch plattform-unabhängige Alternativen ersetzt werden. Beispielsweise können die Arrays während der Modellbildung durch Listen ersetzt werden.

Eine mögliche Erweiterung stellt die Einführung der benutzerdefinierter Einstellungen für Übersetzung einzelner Quellcode-Elemente. Ähnlich wie im Gra2MoL-Einsatz (s. Abschnitt 5.3) kann eine Menge von Mappings definiert werden. Jedes Mapping beschreibt, wie ein bestimmtes Java-Konstrukt in Modellelemente übersetzt wird. Beispielsweise kann durch Mappings gesteuert werden, ob die Blockanweisung einer beliebigen Schleife als Bestandteil des aktuellen Modells übersetzt wird. Eine Alternative dazu wäre eine Schleife, deren Rumpf in einem Untermodell abgebildet wird. Die Mappings könnten zusätzlich eine Bedingung enthalten, die erfüllt sein muss, damit das Mapping relevant wird. Zum Beispiel kann ein Schleifenrumpf, der aus mehr als zwei Anweisungen besteht, als ein Untermodell repräsentiert werden. Sonst werden alle Anweisungen innerhalb der Schleife in das aktuelle Modell eingefügt.

Das Problem der duplizierten Methoden wurde im Rahmen der Masterarbeit durch Code-Anpassungen vor der Modellbildung gelöst. Der dazu nötige Aufwand kann gespart werden, wenn die Erkennung der Duplikationen in den Code-Importer eingebaut wäre. Man könnte während der Modellbildung überprüfen, ob schon ein Modell existiert, welches das gleiche Verhalten hat. Allerdings muss genau definiert werden, wie zwei Methoden bzw. Graphen auf Gleichheit überprüft werden.

Eine wichtige Rolle bei der Modellbildung spielen die Top-Down-Strategien. Benutzerspezifische Strategien müssen als Implementierungen eines Interfaces in Java implementiert werden. Bei komplexen Strategien wäre die modellgetriebene Entwicklung einer passenden Strategie sinnvoll. Der Code-Importer kann um eine Komponente erweitert werden, welche eine Top-Down-Strategie in Form eines jABC-Modells bzw. einer XML-Datei einlesen kann.

Eine weitere Verbesserungsmöglichkeit stellt die automatische Einbindung von Icons bei der Modellgenerierung dar. Die Funktionalität ist im Code-Importer bereits vorhanden und wurde bei der Erstellung der Basic-SIBs benutzt. Zu jedem Basic-SIB wurde vor der Generierung ein Icon erstellt und genau so benannt wie der SIB. So konnte der Code-Importer das richtige Bild finden und dem Modell hinzufügen. Bei QUMA konnte dies nicht wiederholt werden, weil die Anzahl der generierten Modelle variiert hat.

Modellgetriebene Analyse

Die verwendeten Analyse-Metriken zeigen nur einen kleinen Teil des möglichen Potentials. Aufgrund der Nähe von generierten Modelle zum Quellcode können viele andere Software-Metriken auf die Modellebene übertragen werden. Außerdem können die Analysen modellgetrieben durchgeführt werden. Das bedeutet, dass spezielle Modelle zur Berechnung der Metriken entwickelt werden können. Diese Modelle nehmen die Code-Modelle als Eingabe und führen bestimmte Berechnungen auf den Code-Modellen durch. Zusätzlich können modellgetriebene Testszenarien ausgearbeitet werden, mit deren Hilfe die Code-Modelle auf fachliche Fehler geprüft werden. Weitere Modelle könnten die Erstellung von Dokumentation sowie die Visualisierung von Testergebnissen ermöglichen.

Plugins für Modellanpassung

Während des Reengineerings von QUMA wurden die Anpassungen der generierten Modelle manuell im jABC-Editor durchgeführt. Bei den wenigen betroffenen Modellen war der Aufwand noch handhabbar. Für größere Systeme wäre der Einsatz von automatisierten Refactorings auf Modellebene von Vorteil. Bespielsweise könnte automatisch überprüft werden, ob bei der hohen Verschachtelungstiefe einige Teile des Modells in Untermodelle verschoben werden sollen. Hilfreich wäre außerdem ein Mechanismus, der ein Modell in optimale Untermodelle bezüglich der Kontextvariablen zerlegt. Mittels diesem könnte man für einen gegebenen Cluster von Kontextvariablen ein zusammenhängendes Untermodell mit diesen Variablen extrahieren.

Automatisierte Integration

Während des Reengineerings der Java-Anwendung wurde die Integration des generierten Codes manuell ausgeführt. Eine automatisierte Integration ist ebenfalls denkbar. Dabei können die generierten Klassen automatisch als neue Instanzvariablen zum Beispiel mit Hilfe eines Skriptes eingefügt werden. Die Methoden, die durch den Code-Importer in Modelle übersetzt werden, können wie folgt automatisch verändert werden. Im Rumpf der Methoden werden alle Anweisungen gelöscht und durch den Aufruf des entsprechenden Modells ausgetauscht. Alle privaten Methoden, die nicht mehr aufgerufen werden, können entfernt werden, da diese Methoden als Untermodelle repräsentiert werden.

Anpassungen des Code-Generators

Die Auswahl eines geeignetet Code-Generators hat Einfluss auf die Qualität des generierten Codes und die Effizienz des neuen Systems. Der Code-Generator kann neue Projektabhänigkeiten mit sich bringen, die nicht in die Projektstruktur passen. Diese Aspekte müssen bei der Auswahl bzw. Entwicklung des Code-Generators berücksichtigt werden.

Effizienzsteigerung

Durch den Einsatz von sogenannten Zugriff-Services kann die Performance des Systems leicht beeinträchtigt werden. Die Zugriff-Services liefern den Wert einer Instanzvariable pro Aufruf. Wenn in einem Modell mehrere Variablen einer Instanz nacheinander abgefragt werden, können die Werte aller Variablen in einer Hash-Tabelle gespeichert werden.

Verifikation durch Lernverfahren

Im Rahmen dieser Masterarbeit wurde mit Hilfe von Unit-Tests festgestellt, dass sich das Verhalten des Software-Systems durch das Reengineering nicht verändert hat. Die Genauigkeit der Überprüfung hängt dabei von der Testabdeckung ab und kann durch Verwendung eines automatischen Lernverfahrens erhöht werden. Dabei wird das Verhalten des alten Systems durch Anfragen

an das System gelernt. Unter Anfragen werden die den Testszenarien ähnliche Eingaben verstanden. Zu den Anfragen werden die Ergebnisse bzw. die „Antworten“ des Systems gespeichert. Aus der Sequenz Frage-Antwort kann eine Funktion gelernt werden, die das Verhalten des Systems darstellt. Nach dem Reengineering wird die gleiche Sequenz an Anfragen an das neue System gestellt. Zu erwarten ist, dass die Antworten des neuen Systems mit den Antworten des alten System übereinstimmen.

Literaturverzeichnis

[1] Apache POI. http://poi.apache.org/.

[2] Arnold, Robert S.: *Software Reengineering.* IEEE Computer Society Press, 1993.

[3] Arnoldus, Jeroen, Mark G. J. van den Brand, Alexander Serebrenik und Jacob Brunekreef: *Code Generation with Templates.*, Band 1 der Reihe *Atlantis Studies in Computing.* Atlantis Press, 2012.

[4] Baumöl, Ulrike, Jens Borchers, Stefan Eicker, Knut Hildebrand, Reinhard Jung und Franz Lehner: *Einordnung und Terminologie des Software Reengineering.* Informatik Spektrum, 19(4):191–195, 1996.

[5] Beck, Kent: *Extreme Programming.* Pearson Education, 2003.

[6] Bruneliere, Hugo, Jordi Cabot, Frédéric Jouault und Frédéric Madiot: *MoDisco: a generic and extensible framework for model driven reverse engineering.* In: Pecheur, Charles, Jamie Andrews und Elisabetta Di Nitto (Herausgeber): *ASE*, Seiten 173–174. ACM, 2010.

[7] Byrne, E.J.: *A Conceptual Foundation for Software Re-engineering.* In: *Proceedings of the 2013 International Conference on Software Maintanance and Reengineering*, ICSM '92, Seiten 226–235. IEEE Computer Society, 1992.

[8] Canovas, Javier und Jesus Garcia Molina: *Extracting Models from Source Code in Software Modernization.* Software & Systems Modeling, September 2012.

[9] Chikofsky, Elliot J. und James H. Cross II: *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, 7(1):13–17, 1990.

[10] Ebert, Jürgen: *Software-Reengineering - Umgang mit Software-Altlasten.* In: *Informatiktage 2003*, Seiten 24–31, Grasbrunn, 2004. Konradin-Verlag.

[11] Ebert, Jürgen, Bernt Kullbach, Volker Riediger und Andreas Winter: *GUPRO - Generic Understanding of Programs.* Electr. Notes Theor. Comput. Sci., 72(2):47–56, 2002.

[12] Ebert, Jürgen, Volker Riediger und Andreas Winter: *Graph Technology in Reverse Engineering. The TGraph Approach.* In: *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*, Seiten 67–81, 2008.

[13] Eclipse IDE. http://www.eclipse.org/.

[14] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/.

[15] Emrich, Marco und Max Schlee: Codegenerierung mit XFramer und Programmiertechniken für Frames. http://www.neonway.com/codegenerierung-mit-xframer-und-programmiertechniken-fuer-frames.pdf, 2003.

[16] Erdmenger, Uwe, Andreas Fuhr, Axel Herget, Tassilo Horn, Uwe Kaiser, Volker Riediger, Werner Teppe, Marianne Theurer, Denis Uhlig, Andreas Winter, Christian Zillmann und Yvonne Zimmermann: *SOAMIG Project: Model-Driven Software Migration towards Service-Oriented Architectures.* In: Fuhr, Andreas, Wilhelm Hasselbring, Volker Riediger, Magiel Bruntink und Kostas Kontogiannis (Herausgeber):

Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on System Quality and Maintainability (SQM 2011), March 1, 2011, Oldenburg, Satellite Events of the 15th European Conference on Software Maintenance and Reengineering, Band 708, Seiten 15–16, RWTH Aachen, 03 2011. CEUR Workshop Proceedings.

[17] Erich Gamma, Richard Helm, Ralph E. Johnson: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1994.

[18] Ferenc, Rudolf, Árpád Beszédes, Mikko Tarkiainen und Tibor Gyimóthy: *Columbus - Reverse Engineering Tool and Schema for C++.* In: *ICSM*, Seiten 172–181. IEEE Computer Society, 2002.

[19] Fowler, et al: *Refactoring: Improving the Design of Existing Code.* Addison Wesley Professional, 1999.

[20] Fuhr, Andreas: *Model-driven Software Migration into a Service-oriented Architecture: Identifying Model-driven Reverse-Engineering Techniques for SOA Migration.* Formal Sciences Series. AV-Verlag, Saarbrücken and Germany, 2012.

[21] Graphical Modeling Project. http://www.eclipse.org/modeling/gmp/.

[22] *Hibernate.* http://www.hibernate.org/.

[23] iText. http://itextpdf.com/.

[24] Izquierdo, Javier Luis Cánovas und Jesús García Molina: *A Domain Specific Language for Extracting Models in Software Modernization.* In: Paige, Richard F., Alan Hartman und Arend Rensink (Herausgeber): *ECMDA-FA*, Band 5562 der Reihe *Lecture Notes in Computer Science*, Seiten 82–97. Springer, 2009.

[25] *Java-Extractor.* https://github.com/jgralab/java-extractor.

[26] *JavaParser.* https://code.google.com/p/javaparser/.

[27] *JBoss Application Server.* http://www.jboss.org/jbossas/.

[28] Kazman, Rick, Steven Woods und Jeromy Carrière: *Requirements for Integrating Software Architecture and Reengineering Models: CORUM II.* In: *Proc. Working Conf. Reverse Engineering (WCRE)*, Seiten 154–163, Washington, DC, USA, 1998. IEEE Computer Society Press.

[29] *Das Kompetenzcenter Integraler Taktfahrplan NRW (KC ITF NRW).* http://www.kcitf-nrw.de/.

[30] Kienle, Holger M. und Hausi A. Müller: *Rigi - An environment for software reverse engineering, exploration, visualization, and redocumentation.* Sci. Comput. Program., 75(4):247–263, 2010.

[31] Krallmann, Hermann, Annette Bobrik und Olga Levina: *Systemanalyse im Unternehmen - Prozessorientierte Methoden der Wirtschaftsinformatik.* Oldenbourg, Sechste Auflage, 2013.

[32] Margaria, Tiziana und Bernhard Steffen: *Agile IT: Thinking in User-Centric Models.* In: Margaria, Tiziana und Bernhard Steffen (Herausgeber): *ISoLA*, Band 17 der Reihe *Communications in Computer and Information Science*, Seiten 490–502. Springer, 2008.

[33] Margaria, Tiziana und Bernhard Steffen: *Business Process Modelling in the jABC: The One-Thing-Approach.* In: Cardoso, J. und W.M.P. van der Aalst (Herausgeber): *Handbook of Research on Business Process Modeling.* Information Science Publishing, Hershey, PA, USA, 2009.

[34] Margaria, Tiziana und Bernhard Steffen: *Continuous Model-Driven Engineering.* Computer, 42(10):106–109, 2009.

[35] Masak, Dieter: *Legacysoftware: Das lange Leben der Altsysteme.* Springer, 2006.

[36] Mccabe, Thomas J.: *A Complexity Measure.* IEEE Trans. Software Eng., 2(4):308–320, 1976.

[37] McClure, Carma L.: *Software-Automatisierung: reengineering - repository - Wiederverwendbarkeit.* Hanser Verlag, 1993.

[38] Meijler, Theo Dirk, Gert H. Kruithof und Nick van Beest: *Top Down Versus Bottom Up in Service-Oriented Integration: An MDA-Based Solution for Minimizing Technology Coupling.* In: Dan, Asit und Winfried Lamersdorf (Herausgeber): *ICSOC*, Band 4294 der Reihe *Lecture Notes in Computer Science*, Seiten 484–489. Springer, 2006.

[39] Nagel, Ralf: *Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive.* Doktorarbeit, TU Dortmund, 2012.

[40] Naujokat, Stefan, Anna-Lena Lampreche, Bernhard Steffen, S. Jörges und Tiziana Margaria: *Simplicity Principles for Plug-In Development: The jABC Approach.*, 2012.

[41] OMG: *Unified Modeling Language^{TM} (UML®)*, 2011.

[42] Pietrek, Georg und Jens Trompeter (Herausgeber): *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis.* Entwickler.Press, Frankfurt am Main, 2007.

[43] Pérez-Castillo, Ricardo, Ignacio García Rodríguez de Guzmán, Mario Piattini und Christof Ebert: *Reengineering Technologies.* IEEE Software, 28(6):13–17, 2011.

[44] *QUMA.* http://www.quma-nrw.de/.

[45] Rational Rhapsody. http://www-03.ibm.com/software/products/us/en/ratirhapfami.

[46] Rugaber, Spencer und Kurt Stirewalt: *Model-Driven Reverse Engineering.* IEEE Software, 21(4):45–53, 2004.

[47] Scalise, Eugenio P., Jean-Marie Favre und Nancy Zambrano: *Model-Driven Reverse Engineering and Program Comprehension: an Example.* Ingeniare. Revista chilena de ingeniería, 18(1):76–83, 2010.

[48] Schulze, Martin: *Modellierung von LOTOS-Spezifikationen durch TGraphen.* Technischer Bericht S 367, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1995.

[49] Seacord, Robert C., Daniel Plakosh und Grace A. Lewis: *Modernizing Legacy Systems - Software Technologies, Engineering Processes, and Business Practices.* SEI series in software engineering. Addison-Wesley, 2003.

[50] Sneed, Harry M.: *Aufwandsschätzung von Software-Reengineering-Projekten.* Wirtschaftsinformatik, 45(6):611–620, 2003.

[51] Sneed, Harry M., Ellen Wolf und Heidi Heilmann: *Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung.* dpunkt Verlag, 2010.

[52] Soley, R.: *Extracting UML from legacy applications.* SOA Web Services Journal, Oktober 2006.

[53] Sommerville, Ian: *Software Engineering.* Pearson Education, 2011.

[54] Stahl, Thomas, Markus Völter, Sven Efftinge und Arno Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* Dpunkt Verlag, 2007.

[55] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges und Christian Kubczak: *Model-Driven Development with the jABC.* In: Bin, Eyal, Avi Ziv und Shmuel Ur (Herausgeber): *Haifa Verification Conference*, Band 4383 der Reihe *Lecture Notes in Computer Science*, Seiten 92–108. Springer, 2006.

[56] Tidwell, Doug: *XSLT - mastering XML transformations.* O'Reilly, 2001.

[57] Apache Velocity. http://velocity.apache.org/.

[58] Vogel, Oliver, Ingo Arnold, Arif Chughtai, Edmund Ihler, Timo Kehrer, Uwe Mehlig und Uwe Zdun: *Software-Architektur - Grundlagen, Konzepte, Praxis (2. Aufl.).* Spektrum Akademischer Verlag, 2009.

[59] *Verkehrsverbund Rhein-Ruhr.* http://www.vrr.de/.

[60] Völter, Markus und Thomas Stahl: *Architektur und Generierung.* iX, (1), 2003.

[61] Wagner, Christian: *Modellgetriebene Software-Migration.* Doktorarbeit, Universität Potsdam, 2012.

[62] Weil, D.: *Java EE 6: Enterprise-Anwendungsentwicklung leicht gemacht.* Software + Support, 2012.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABC	Application Building Center
ADM	Architecture-Driven Modernization
API	Application Programming Interface
AST	Abstract Syntax Tree
DAO	Data Access Object
EMF	Eclipse Modeling Framework
GMP	Graphical Modeling Project
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
J2EE	Java Platform, Enterprise Edition
jABC	Java Application Building Center
MDSD	Model-Driven Software Development
MRI	Magnetic Resonance Imaging
NRW	Nordrhein-Westfalen
OO	objektorientiert
SIB	Service Independent Building Blocks
SLG	Service Logic Graphen
SPNV	Schienenpersonennahverkehr
QUMA	Qualitätsmanagement im Personennahverkehr
VRR	Verkehrsverbund Rhein-Ruhr
XMDD	Extreme Model-Driven Design
XML	Extensible Markup Language
UML	Unified Modeling Language

Anhang

A Basic SIBs

In folgender Tabelle werden alle im Rahmen der Masterarbeit verwendeten Basic-SIBs aufgelistet. Für jeden SIB wird unter anderem angegeben, ob dieser aus dem jABC4-Plugin übernommen oder mithilfe des Code-Importers erstellt wurde. Die arithmetischen Operatoren wurden für die Typen `Integer`, `Float`, `Double`, `Short` implementiert.

Name	**Beschreibung**	**Übernommen**	**Erstellt**
Cast	Implementiert den Cast-Operator	✓	
ConcatStrings	SIB zur Konkatenation zweier Strings	✓	
IsEmpty	Überprüft, ob eine Kollektion leer ist	✓	
IsInstance	Implementiert den `instanceof`-Operator	✓	
Iterate	Iteriert über eine Kollektion und gibt entweder das nächste Element im Branch *next* oder *exit* zurück	✓	
PutToContext	Ordnet einen Wert einer Kontextvariable zu	✓	
toString	Konvertiert ein Objekt in String	✓	
And, Or, Not, Xor	SIBs für boolesche Operationen	✓	
Equals	Vergleicht zwei Objekte und gibt *yes* aus, wenn diese gleich sind. Sonst wird *no* ausgegeben.	✓	
NotEquals	Vergleicht zwei Objekte und gibt *no* aus, wenn diese gleich sind. Sonst wird *yes* ausgegeben.		✓
Less	Implementiert den <-Operator	✓	
LessEquals	Implementiert den =<-Operator	✓	
Greater	Implementiert den >-Operator	✓	
GreaterEquals	Implementiert den >=-Operator	✓	
BooleanValue	Wertet eine boolesche Variable aus und gibt *yes* zurück, wenn die Variable den Wert *true* besitzt. Sonst wird *no* ausgegeben		✓
Add_<Typ>	Implementiert den +-Operator. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Substract_<Typ>	Implementiert den −-Operator. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Multiply_<Typ>	Implementiert den ∗-Operator. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Divide_<Typ>	Implementiert den /-Operator. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓

Modulo_<Typ>	Implementiert den %-Operator. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Negative_<Typ>	Implementiert den Operator (−). Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Positiv_<Typ>	Implementiert den Operator(+). Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Increment_<Typ>	Implementiert den Operator ++. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
Decrement_<Typ>	Implementiert den Operator −−. Das Ergebnis wird in eine Kontextvariable Typ <Typ> geschrieben.		✓
GetInstanceField	Liest den Wert der Instanzvariable mit gegebenen Namen aus der gegebenen Instanz		✓
GetStaticField	Liest den Wert der Klassenvariable mit gegebenen Namen aus der gegebenen Klasse		✓
ArrayLength	Implementiert den Ausdruck `array.length` für den gegebenen Array.		✓
NewArrayInstance	Erzeugt eine neue Array-Instance vom gegebenen Typ und Länge.		✓
GetArrayElement	Liest den Wert eines Array-Elements der gegebenen Position.		✓
SetArrayElement	Schrebt einen neuen Wert in die gegebenene Position		✓

B Generierter Code

Im Folgenden befindet sich die Java-Klasse, die zum Modell „Linien-Pünktlichkeit“ aus Abschnitt 4.3.4, mithilfe des Code-Generators *jabc4-codegen-java* erstellt wurde. Einige Implementierungsdetails wegen besserer Übersichtlichkeit entfernt.

```
package de.vrr.quma.spnv.auswertung.export;

import ...

public final class LinienPuenktlichkeitSheetCreator__createSheet implements
    SLG {
  // adjacency map (from id to branch to id).
  private final Map<String, Map<String, String>> _adjacencyMap_;
  // map from id to container.
  private final Map<String, SIB> _idToContainer_;
  // first sib to execute.
  private SIB _startSIB_;

  public LinienPuenktlichkeitSheetCreator__createSheet() {
    _adjacencyMap_ = new HashMap<String, Map<String, String>>();
    _idToContainer_ = new HashMap<String, SIB>();
    // adjacency for 'new LinienPuenktlichekeitLabelHolder'.
    final Map<String, String> adjacency = new HashMap<String, String>();
    adjacency.put("success", "f00f8baf_5492_4286_8364_2d555c6ee549");
    _adjacencyMap_.put("34c51bf3_5f75_49e2_87a2_c62d879d8c7d", adjacency);
    // adjacency for 'SheetCreatorServices__createNewSheet'.
    ...
    // adjacency for 'Sheet#setDefaultColumnWidth'.
    ...
    // adjacency for 'LinienPuenktlichkeitSheetCreator__createHeaders'.
    ...
    // adjacency for 'LinienPuenktlichkeitSheetCreator__createContent'.
    ...
    // adjacency for 'AbstractSheetCreator__autosizeColumns'.
    ...
    // adjacency for 'success'.
    ...
    // initialize start sib.
    _startSIB_ = _idToContainer_
        .get("34c51bf3_5f75_49e2_87a2_c62d879d8c7d");
  }

  ...

  public String execute() {
    // execute the graph
    SIB _currSIB_ = _startSIB_;
    String _branch_ = null;
    while (_currSIB_ != null) {
      _branch_ = _currSIB_.execute();
      if (_currSIB_.isEndSIB()) {
        break;
      }
      final String _succId_ = _adjacencyMap_.get(_currSIB_.getId()).get(
          _branch_);
      if (_succId_ == null) {
        throw new IllegalStateException("SIB '"
            + _currSIB_.getDisplayName()
            + "' returned the branch '" + _branch_ + "'"
```

```
            + ", which has no successor.");
        }
        _currSIB_ = _idToContainer_.get(_succId_);
      }
      return _branch_;
    }

    public SuccessReturn getSuccessReturn() {
      return new SuccessReturn();
    }

    public class SuccessReturn {
    }

    ...

    // sib containers.
    // container for service call 'new LinienPuenktlichekeitLabelHolder'.
    private class SIB34c51bf3_5f75_49e2_87a2_c62d879d8c7d extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for graph abstraction 'SheetCreatorServices__createNewSheet
        '.
    private class SIBf00f8baf_5492_4286_8364_2d555c6ee549 extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for service call 'Sheet#setDefaultColumnWidth'.
    private class SIBf7d97834_ac5f_46e3_b268_f27a6d3bd98a extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for graph abstraction
    // 'LinienPuenktlichkeitSheetCreator__createHeaders'.
    private class SIBabd6a131_e437_42f1_89e8_d9dcba64fffa extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for graph 'LinienPuenktlichkeitSheetCreator__createContent'.
    private class SIBa5e451ba_6719_475b_9858_03fcd7aed465 extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for graph abstraction 'AbstractSheetCreator__autosizeColumns
        '.
    private class SIB92246bc8_f814_4c58_94d6_61100672ba0a extends
        _AbstractSIBContainer_ {
        ...
    }
    // container for graph i/o 'success'.
    private class SIBf53a5a8f_4af2_4f37_8124_5cd01e70a7e9 extends
        _AbstractSIBContainer_ {
        ...
    }
}
```

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift